

**INFORMATION LEAKAGE  
IN ENCRYPTED IP VIDEO TRAFFIC**

A Thesis  
Presented to  
The Academic Faculty

By

Christopher Wampler

In Partial Fulfillment  
Of the Requirements for the Degree  
Masters of Science in the  
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2014

Copyright © Christopher Wampler 2014

# INFORMATION LEAKAGE IN ENCRYPTED IP VIDEO TRAFFIC

Approved by:

Dr. Raheem Beyah, Advisor  
School of Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. John Copeland  
School of Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. Ghassan AlRegib  
School of Electrical and Computer Engineering  
*Georgia Institute of Technology*

Date Approved: 4 November 2014

*Dedicated to my wife Laura*

*Thank you for your love and patience*

## ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Beyah for his ideas and suggestions throughout my time working on this research. He was always able to give me perspective and keep me from straying too far from the most important objectives. I thank my thesis committee for reading and providing feedback on my writing as well as Dr. Selcuk Uluagac for keeping track of my progress and providing his feedback.

I attribute my understanding of video processing, and a great many other things, to Greg Finn and express my appreciation for his mentoring and everything I have been able to learn from him.

I have been indirectly funded in this research by Sandia National Laboratories who has covered tuition for my degree program and provided the opportunity to conduct this research.

Finally, I express my appreciation to my four children Audrey Jane, Megan, Ben and Simon for giving up time with their father to be able to finish my schooling and to my wife Laura for her hours of coaching, proof reading (all commas in the document are attributed to her), and taking care of everything else in our family so I could complete this thesis.

# TABLE OF CONTENTS

<b>ACKNOWLEDGEMENTS</b> . . . . .	<b>iv</b>
<b>LIST OF TABLES</b> . . . . .	<b>vii</b>
<b>LIST OF FIGURES</b> . . . . .	<b>viii</b>
<b>SUMMARY</b> . . . . .	<b>ix</b>
<b>I INTRODUCTION</b> . . . . .	<b>1</b>
<b>II BACKGROUND</b> . . . . .	<b>3</b>
2.1 Video Encoding . . . . .	3
<b>III RELATED WORK</b> . . . . .	<b>8</b>
3.1 Phrase Detection in Voice Over IP . . . . .	8
3.2 Side-channel Attacks . . . . .	9
<b>IV INFORMATION LEAKAGE IN ENCRYPTED IP VIDEO TRAFFIC</b>	<b>11</b>
4.1 Leakage Factors Throughout the Video Pipeline . . . . .	11
4.1.1 Image Content . . . . .	13
4.1.2 Video Camera . . . . .	14
4.1.3 Encoding Application . . . . .	15
4.1.4 Encryption . . . . .	16
4.1.5 Computing Platform . . . . .	17
4.1.6 Network Transmission . . . . .	18
4.1.7 Traffic Analysis . . . . .	19
4.2 Initial Investigation . . . . .	20
4.2.1 Automated Test Environment . . . . .	20
4.2.2 Event Detection Across Multiple Codecs . . . . .	22
4.3 Encoder Timing Analysis of x264 . . . . .	28
4.3.1 Algorithm Time-stamping . . . . .	28
4.3.2 Analysis of Timing Results . . . . .	30
4.4 Skype's H.264 Encoder . . . . .	32
4.4.1 Traffic Analysis in a Laboratory Environment . . . . .	33
4.4.2 Event Type Classification . . . . .	35

4.5	Collecting Outside User Video Calls . . . . .	37
4.5.1	Video Capture Collection Framework . . . . .	37
4.5.2	Overview of Collected Data from Outside Sources . . . . .	44
<b>V</b>	<b>CONCLUSION AND FUTURE WORK . . . . .</b>	<b>48</b>
<b>APPENDIX A</b>	<b>— CODE LISTINGS FOR SKYPE AUTO-ANSWER .</b>	<b>49</b>
<b>APPENDIX B</b>	<b>— ADDITIONAL TRAFFIC ANALYSIS PLOTS . . . .</b>	<b>61</b>
<b>REFERENCES</b>	<b>. . . . .</b>	<b>68</b>

## LIST OF TABLES

1	Considerations for information leakage detection in the video encoding pipeline.	12
2	Test cases and detection methods. . . . .	27
3	Order of events for captures . . . . .	61

## LIST OF FIGURES

1	Lossy compression examples 80%, 84%, and 87% compressed, left to right.	5
2	Temporal compression content sources for I, P, and B frames. . . . .	6
3	Vulnerability of ECB versus CBC encryption to information leakage. . . . .	9
4	Video transmission pipeline. . . . .	11
5	Light box for controlled test environment. . . . .	21
6	Skype network capture marking time of object passed in front of camera. . .	24
7	D-Link network capture marking time when lights are turned off then on again.	25
8	GStreamer pipeline construction. . . . .	29
9	Open source x264 encoder performance profiling. . . . .	30
10	Laboratory environment captures showing repeatability of measurements. . .	33
11	K-means classification of traffic analysis for laboratory video captures. . . .	36
12	Excerpts from instruction video shown to participants during calls. . . . .	38
13	Control structure for Skype Auto-Answer application. . . . .	39
14	Web page showing participant survey and summary of research. . . . .	41
15	Web page showing participant instructions for conducting video calls. . . .	42
15	(continued). . . . .	43
16	Server desktop view of an incoming video chat recording session. . . . .	44
17	Locations of U.S. callers participating in experiment. . . . .	45
18	Variation in bandwidth versus inter-arrival time for collected data. . . . .	45
19	User captures with average bandwidths of 139, 106, and 77 KBps. . . . .	46
20	User captures with average bandwidths of 19, 11, and 9 KBps. . . . .	47
21	Control and data flow for Skype Auto-Answer application. . . . .	49
22	Test script for video loop-back feature used with call instructions video. . .	50
23	Captures from Google Hangouts and WebRTC. . . . .	62
24	Captures from D-Link camera with varying resolution. . . . .	63
25	Captures from D-Link camera with varying frame rates. . . . .	64
26	Captures from Dell, Samsung, and Aspire devices with various cameras. . .	65
27	Captures from Acer laptop running Ubuntu 12.04 and Windows 7. . . . .	66
28	Captures with varying levels of bandwidth limitation. . . . .	67



## SUMMARY

We show that information leakage occurs in video over IP traffic, including for encrypted payloads. It is possible to detect events occurring in the field of view of a camera streaming live video through analysis of network traffic metadata including arrival time between packets, packets sizes, and video stream bandwidth. Event detection through metadata analysis is possible even when common encryption techniques are applied to the video stream such as SSL or AES. We have observed information leakage across multiple codes and cameras. Through timestamps added to the x264 codec, we establish a basis for detectability of events via packet timing. Laboratory experiments confirm that this event detection is possible in practice and repeatable. By collecting network traffic captures from over 100 Skype video calls we are able to see the impact of this information leakage under a variety of conditions.

# CHAPTER I

## INTRODUCTION

We have been able to demonstrate that information leakage occurs in encrypted video over IP traffic. Across a range of codecs tested, we observed that through network traffic analysis it is possible to detect events occurring in view of a camera streaming live video. This work is similar in concept to voice over IP work by Wright et al. [33], who detected spoken phrases through analysis of audio network traffic. An original investigation into this area was conducted by Suriyanarayanan [29] which established that events could be detected through analysis of video network traffic. We expand on this previous work by establishing the theoretical basis for information leakage in streaming video and performing several experiments to quantify what types of events are detectable.

We found that variations in packet size and arrival time could indicate activity in a video stream and tested what types of events are detectable with a variety of codecs, cameras, and processing hardware. We focus particularly on the H.264 [1] codec both in the Skype peer-to-peer video chat network [2] [12] and the open source x264 video encoder [9] implementations. We demonstrate the connection between varying encode times based on events being encoded and the resulting variation in packet arrival times by collecting timing measurements from the x264 encoder. We then establish the repeatability and consistency of measurements for a single camera and computer in a laboratory environment. With consistent results in a laboratory environment we are able to algorithmically detect events. We have attempted to classify events by type using the k-means auto classification algorithm but have thus far been unsuccessful. We repeated our laboratory experiments in a broader environment by recruiting Skype users to call our server and record a specific sequence of activities. We were able to observe a much wider variety of network traffic measurements in this manner. Our preliminary manual analysis of these recorded sessions showed that in over 70% of captures some event was discernible, with much higher probability in high

bandwidth connections.

In Chapter 1, we introduce the work and give a summary of findings. In Chapter 2, we give a background of relevant information for video capture and encoding. We discuss related work in Chapter 3, covering information leakage in VoIP and side channel attacks. Chapter 4 forms the main body of this document. Section 4.1 discusses the theoretical basis of information leakage throughout the video pipeline. Our original tests showing event detection in a variety of codecs are documented in Section 4.2. We give evidence for our encoder performance theory through time stamps added within x264 encoder in Section 4.3. In Section 4.4, we demonstrate repeatability of event detection in a laboratory environment with Skype’s H.264 codec. We extend our experiments outside the laboratory in Section 4.5 by collecting and analyzing video traffic captures from other Skype users. Finally, in Chapter 5 we conclude with the results of our research and a discussion of future work.

## CHAPTER II

### BACKGROUND

#### *2.1 Video Encoding*

To appreciate the context of this work, some understanding of video encoding and image processing is necessary. This section gives some of the video encoding background relevant to this research. A short overview of video compression is given in [27]. More detailed information about video encoding can be obtained from [23].

Video encoding begins with raw image data from a camera's photo sensor array. Photons reflected from a subject strike a sensor element and produce an analog electrical response corresponding to the wavelength and intensity of the light. This analog output is digitized as an array of pixels in a particular data format. The number of individual sensing elements in a camera's photo sensor array determines the maximum effective resolution of the final image. A small sensor array, with relatively few photo-sensing circuits, will result in a low quality image. A large sensor, with a high density of quality photo-sensing circuits, makes it possible to capture detailed high resolution images. The electrical and photo-reactive characteristics of the sensor materials and the reaction speed of the circuitry will determine how much time is needed to gather enough light to make a distinct measurement and therefore the maximum rate at which new, high-quality, images can be produced from that camera. A still camera will produce a single digitized representation of the light seen through its aperture while a video camera will produce a stream of such images.

There are many data formats which can be used to represent a captured image. The classic RGB format uses one byte each to specify the intensity of red, green and blue light for each pixel. YUV format instead represents color as luminance and chrominance and versions of YUV specify how many bits are used to compose a pixel in that format [24]. For example, the 442 in YUV-442 corresponds with 4 bits used to represent the luminance, and 6 bits for chrominance (Cb, Cr). There are 4 bits Cb, and 2 bits Cr which together represent

coordinates in a two dimensional color space [23]. Other representations (e.g., YUV444, YUV420, YUV12) may be used based on what will produce an image which yields a quality sufficient for the particular application. Using fewer bits to represent each pixel will result in a smaller image size but also a less precise representation of color. For example, YUV12 which uses 1 bit for luminance, 1 for Cb, and 2 for Cr, will only use 4 bits per pixel rather than the 10 bits per pixel required for YUV-442 but will have a much more limited range of colors that can be represented by that format. YUV pixel formatting is often preferred over the arguably more direct RGB color format because it can take human color perception into account, allowing smaller data sizes tailored to what the human eye can actually discern.

Once a raw image is captured, it undergoes compression and format changes to match the standard of the particular codec used. In our work, we are focused primarily on the H.264 codec which is used in the Skype video chat client, and the MJPEG codec, which is used widely in security applications. We also give some consideration to the VP8 [20] codec, the now open source codec used for Google Hangouts. One of the oldest and simplest codecs is the MJPEG (motion JPEG) codec which takes a series of JPEG [31] compressed images and displays them in sequence to produce a motion picture [23].

Compression techniques in general convert a large data set to a smaller data set by finding patterns and repeated information [32]. The original data can then be represented in a smaller size expressed in terms of those patterns. An image that has many patterns is considered to have low entropy (i.e., high redundancy) and will be very compressible. Conversely, an image with high entropy (i.e., low redundancy), will contain fewer patterns resulting in less compressibility. An image with high entropy is often said to contain a lot of high frequency information.

In image processing applications, it is possible that some of the high frequency information may be ignored with minimal impact on the usability of the compressed file. Compression which drops some high frequency data is considered lossy compression [10]. A compression process that allows the original data to be recovered exactly from the compressed file is called lossless compression. This is the type of compression used for text or other files where no high level information about the use of the data is known by the

compression algorithm, and loss of any information would be considered a failure in the algorithm. The high level information that allows a lossy image compression algorithm to drop some data is the fact that humans can only see so much detail in an image.

JPEG uses a combination of lossless and lossy compression. JPEG lossy image compression takes advantage of the fact that the file to be compressed is an image and there are details in an image that the human eye cannot perceive. The brain will also fill in details that it knows should be in a particular image even when the actual detail is not there. This is what allows us to recognize someone from a blurry photograph.



Figure 1: Lossy compression examples 80%, 84%, and 87% compressed, left to right.

Figure 1 demonstrates the effects of lossy compression on an image which is 563 KB uncompressed. At 80% compressed (112.3 KB), the image remains sharp, retaining most of its original detail. This is achieved by removing some high frequency information content of the image that is not even seen in a casual glance. The pixels are not actually deleted, but rather are changed to match adjacent pixels. This results in more similar pixels throughout the image decreasing the overall entropy. The image is then compressed using lossless techniques to achieve the final compression level. The center panel, at 84% compression (88.9 KB), shows that some detail has been lost in the background grass, in the hair, and around the eyes due to lossy image processing. In the rightmost panel there has been significant loss of high frequency detail which has resulted in color distortion. This reduction in detail allows the image to be compressed an additional 15.7 KB to 87% compressed (73.2

KB), but the loss of detail is too extreme to be useful in general. A balance is necessary with lossy compression to achieve a small output size while still producing an image quality that is acceptable and useful to the end user.

Lossy and lossless compression are both spatial compression techniques which take advantage of the similarities within a single still image. For video, we can also take advantage of temporal compression, a technique used by the H.264 and VP8 codecs. Because individual video frames are captured many times per second (30 frames per second is common), adjacent images or frames will contain much of the same information. For this reason, a video sequence may be broken up into a series of I, P, and B-frames where different strategies of temporal compression are used.

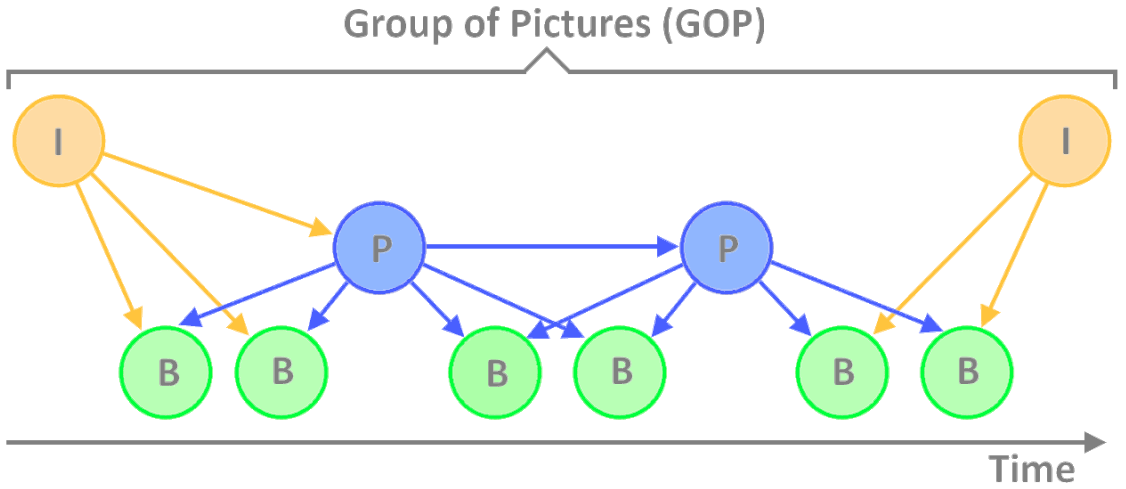


Figure 2: Temporal compression content sources for I, P, and B frames.

Figure 2 shows how temporal compression [13] is performed for each frame type. An I-frame (Intra-coded-picture) does not take advantage of any temporal compression and employs only spatial compression techniques. MJPEG could be said to contain only I-frames. Since other frame types are based on each other's data, I-frames are necessary in an environment which experiences data loss to ensure that after a loss the codec will be able to recover. A P-frame (Predicted-picture) compares its content with the previous I or P frame and represents only the differences in image content compared to its reference frame. A P-frame could be considered a compressed delta of the differences between two

frames. A B-frame (Bi-predictive-picture) utilizes comparison with both the previous and subsequent I and P frames respectively. It represents its frame as a delta from the source frame that is most similar to that portion of the frame. These P and B frames are what give newer codecs such as H.264 and VP8 a significant reduction in output size.

The set of frames from one I-frame to the next is referred to as a group of pictures (GOP). In Figure 2, the GOP size is nine which counts the P and B frames between two I-frames and the preceding I-frame. Keeping the GOP size small is a concern when information may be lost during transmission. If any portion of an I-frame is lost, subsequent P and B frames which contain deltas from the I-frame continue to visually propagate errors from the loss of the single I-frame across multiple subsequent frames until a new I-frame is transmitted, starting a new GOP. A smaller GOP size then results in a more reliable transmission with few artifacts (pixels that are rendered incorrectly due to information loss), but also takes less advantage of the excellent compression available from P and B frames.

Moreover, the computation needed to find an optimal set of pixels to compare between frames is complex and time consuming, but the resulting output data format (including P and B frames) is relatively simple. This allows the decoder on the receiving end to be constructed much more cheaply. This front-loading of work in the video pipeline with a complex encoder and simple cheap decoder has several benefits. For a professional recording or live broadcast, the studio can use high quality equipment to ensure that the video produced has been encoded in the best way possible to represent their footage. The consumer of the video can then use a simple decoder in their television, computer, or hand-held device making it easier to afford to access the video content. Also, by front-loading the work of encoding the resulting compressed data is much smaller which makes for easier transmission of the data whether wirelessly or when stored to media such as a DVD or Blu-ray.

While many of these video encoding decisions are practical, we will show that these features, together with current encryption methods, result in information leakage in transmitted video streams. We show the presence of this information leakage through network traffic analysis of common encoding applications in Section 4.2.1 and demonstrate its origins in encoder algorithm implementation in Section 4.3.



## CHAPTER III

### RELATED WORK

#### *3.1 Phrase Detection in Voice Over IP*

Our work focuses on side-channel attacks against streaming video technologies. Related research by Wright et al. [33] investigates the ability to decode spoken phrases from voice over IP (VoIP) traffic. Video over IP transmissions are similar to VoIP in that IP traffic is expected to traverse untrusted domains. For this reason video and voice only chats are both encrypted in an attempt to prevent eavesdropping by an adversary. It was found, however, that due to the nature of the spoken word and common audio encoding techniques, current VoIP encryption practices are insufficient [33].

Common voice codecs are based on code-excited linear prediction which has been shown to produce predictably sized packets based on the input audio stream. Specifically, spoken words can be broken down into the individual sounds uttered to construct them. These component sounds are called phonemes and there are only about 50 such phonemes in the English language. This taken together with the similarity of packet sizes for each specific encoded phoneme gives a strong starting point for decoding speech through network traffic analysis [33]. Wright et al. shows that through their techniques, it is possible to identify a specific spoken phrase from encrypted VoIP traffic with accuracies exceeding 50%. Due to the differences in video compared to voice traffic, our work focuses on identifying what the phoneme equivalent would be in video over IP traffic. Through analysis of network traffic metadata such as packet size, inter-arrival time, and overall stream bandwidth, we can see that information is being leaked. Similar to VoIP, the commonly used encryption techniques for video over IP do not pad or attempt to obfuscate the size or timing of their input data leaving all of these metrics available for exploitation by an adversary.

### 3.2 Side-channel Attacks

Side-channel attacks exploit predictability in the operation or output of cryptographic algorithms. This predictability allows an adversary to learn something about the plain text or key without actually deciphering it; this is information leakage. Figure 3 demonstrates that while a cryptographic algorithm may be strong in theory, the way that algorithm is employed may allow information leakage to occur. In this case the strong AES algorithm used in electronic code book (ECB) mode for image encryption allows a side-channel attack to occur. An attacker does not need to decrypt the center image in order to determine what was originally encrypted. Through patterns in the cipher text, the overall message of the original images becomes apparent. The right panel, which uses the improved cipher-block chaining (CBC) mode [14], shows that the same frequency analysis attack that was successful in ECB mode can no longer occur.

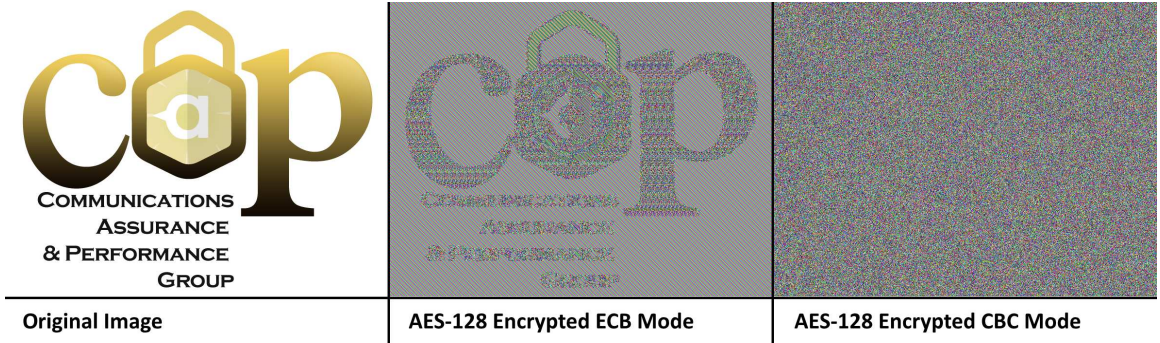


Figure 3: Vulnerability of ECB versus CBC encryption to information leakage.

A side-channel attack does not focus on breaking strong cryptographic algorithms, but instead focuses on analyzing side effects of secure processes to gather information through alternative methods. Early side-channel attacks allowed an eavesdropper to monitoring electromagnetic emmissions from a computer monitor to determine what was being displayed [30]. Recently it has been demonstrated that using an iPhone’s accelerometer, an attacker can determine passwords being entered on a nearby keyboard [26].

Other side-channel attacks exploit the side effects of the encryption process and have even been used for encryption key retrieval. For example, a timing or power monitoring attack takes advantage of the differing complexity required to process a zero or one bit

in a key. Variations in power consumption and computation time result from an effort to optimize performance in these areas [16]. Most optimizations in hardware circuitry and software algorithms which allow an operation to complete more quickly, when possible, are preferable. This results in higher throughput and lower power consumption. However, when the difference in processing time and power use for processing a zero bit in an encryption key is significantly different than for processing a one bit, those optimizations can be taken advantage of to reveal the bits contained in the key itself. Kocher [25] demonstrated that this type of was possible against RSA decryption keys. Brumley [16] extended this work showing that timing attacks can be executed remotely.

Our work does not attempt to reveal the original content of a transmitted video but instead uses network traffic analysis to show that there is a predictable response in the network traffic when events occur in view of a camera transmitting live video. Chen et al. [17] discusses the increasing number of web applications vulnerable to side channel analysis and discusses that effective and efficient mitigations will have to be tailored to each specific application. Dyer et al. [18] discusses methods used to prevent identification of websites visited by a user through network traffic analysis. Their work stipulates that that there are no efficient countermeasures to side-channel traffic analysis for website identification. We demonstrate that side-channel attacks are possible against live video over IP transmissions and show what types of analysis reveal information about the encrypted content.

## CHAPTER IV

### INFORMATION LEAKAGE IN ENCRYPTED IP VIDEO TRAFFIC

#### 4.1 *Leakage Factors Throughout the Video Pipeline*

Throughout our investigation, we have identified a number of factors which we believe affect the detection of recorded events through network traffic analysis (NTA). This list of contributing elements is given in Table 1. Each of the items listed in Table 1 fit into a stage of the video encoding pipeline as represented in Figure 4.

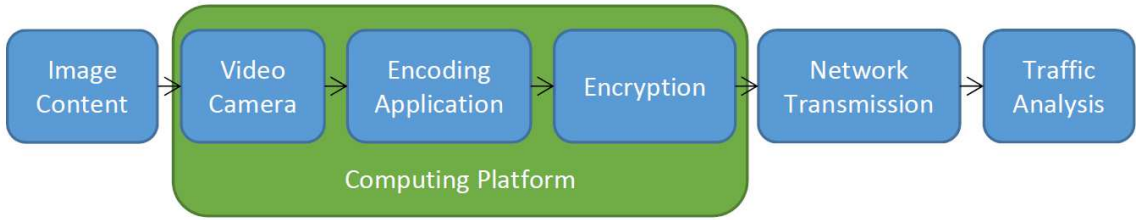


Figure 4: Video transmission pipeline.

This figure is useful in conceptualizing the video encoding process, but it is not a precise representation of how encoders are actually implemented. We represent the general flow of information from image capture by a video camera, through network transmission, packet interception and analysis of network traffic. There is no representation of the video decode and display process which would occur after reception of network packets by the destination platform. Traffic analysis can occur regardless of whether or not the video is ever received and decoded at its final destination. Some steps in the pipeline such as encoding and encryption may also be interleaved in a real world system, but we can still think of the progression of data through each step as indicated here.

Each of the following sub-sections elaborates on the items in Table 1 and illustrates how each item could contribute to information leakage through NTA.

Table 1: Considerations for information leakage detection in the video encoding pipeline.

<b>Image Content</b>	
Image complexity and compressibility	Speed of moving object
Light to dark transition	Moving object contrast with background
Percentage of frame changing	Direction of motion
Still to moving scene or object	
<b>Video Camera</b>	
Image capture resolution	Sensor quality
Focus optics	Frame rate
<b>Encoding Application</b>	
Encoding standard (H.264, MJPEG, VP8)	Resolution
Implementation of encoding algorithm	Compression effort
Encoder settings	Output data rate target
Frame rate	Effects when encoder starts and stops
Constant vs. variable frame rate	
<b>Encryption</b>	
Encrypted vs. unencrypted data	Portion encrypted:
Type of encryption (AES, SSL, VPN)	- Payload only
Encrypted tunnel	- Payload and video headers
Padded data	
<b>Computing Platform</b>	
Processor speed	Other concurrent applications
System memory	Camera interface (USB, PCI, embedded)
Operating system scheduling	
<b>Network Transmission</b>	
Protocol type used (TCP, UDP, RTP)	Network bandwidth
Type of network:	Jitter
- LAN	Varying route through network
- Wi-Fi	Contending traffic
- Internet	Traffic tap point
- Mobile	
<b>Traffic Analysis</b>	
Packet filtering	Statistical analysis methods:
Source and destination IP address	- Moving average
Source and destination port	- Standard deviation
Inter-arrival time of packets	- Statistical distributions of metrics
Packet size	- Frequency analysis
Overall video stream bandwidth	Time window length for data averaging
The number of packets received	

#### 4.1.1 Image Content

The complexity of the captured image is reflected in the network traffic by the bandwidth of the encoded data. Complex images are less compressible, and due to theoretical limits on the compressibility of data, they will require more data to be transmitted. As a result of temporal compression techniques used in codecs like H.264, a still scene will be highly compressible. Even if the content of that scene is very complex, if it is not changing, it can be represented as a repetition of already transmitted data, an essentially empty P-frame. Moving objects in a scene prevent the simple situation of repeating the previous frame and require additional data to be transmitted and additional computation time to find the set of pixels on which to base the new changes. This implies that motion can result in both a bandwidth change as well as a packet arrival time delay.

Similarly, a bright to dim lighting transition has the effect of changing, even if subtly, every pixel in the represented image. This large change across the entire scene results in a large amount of new data to be sent across the network. Moreover, a bright to very dark lighting transition has the effect of changing the amount of detail that is visible in the texture of the scene, causing a change in the complexity of the image in addition to a reduction in the range of perceived colors. In the reverse the same is true, in a very dark scene where image complexity is hidden until lighting increases, the compressibility of the image is greatly impacted. These types of change have an effect on both temporal and spatial compression and therefore are more likely to appear across codecs.

Other factors impacting motion detection include the speed and size of a moving object. If the object moving only effects a small percentage of the image space it will have a correspondingly small change in the network traffic and may be lost below the noise floor. Also an object moving slowly through a high frame rate capture can be represented as a small change set between frames. An object moving very quickly through a low frame rate capture may be missed altogether or may affect only a small number of frames, making it less observable in network traffic variations. In a similar fashion, the direction of motion might also be detected based on variations in the size of individual packets representing portions of a frame, but no obvious indicators have been observed.

### 4.1.2 Video Camera

The capabilities of the capture device will also have an effect on the character of the network traffic. Because the capture device will not change during transmission, however, whatever effect is seen will be observed throughout the entire video stream. Video camera characteristics may cause events to be more or less easily observable in NTA.

The resolution of the captured image gives a baseline for video bandwidth. However, other factors will also have an effect on this baseline such as the focus and resulting sharpness of an image. A high resolution image in good focus will capture more complex surface textures thereby resulting in a more complex image requiring higher bandwidth. Also, more time is required to compute changes for moving objects with detailed texture, possibly resulting in longer packet arrival times. An out of focus capture blurs image details and decreases data complexity and therefore network traffic responsiveness to observed activity.

Similar to the optical focus of an image, the camera sensor quality will have an effect on image complexity independent from the output resolution of the camera. A digital camera has an array of photo sensors which each sample the wavelength and intensity of arriving photons as an electrical signal. If the number of pixels in the digital image output is much larger than the number of individual elements in the sensor array, then those pixels will be some interpolation of color values which creates an effect similar to image blurring. This results again in less detailed textures and a less complex image transmission.

Each camera will have a maximum effective frame rate it can operate at based on its sensor technology. Each photo sensor in the array must be read and reset before the next capture with sufficient time given for the sensor output to stabilize before each reading. An attempt to encode a video at 20 frames per second using a camera which is only capable of refreshing five times per second would result in the same data being transmitted four times. With temporal compression this results in empty change sets.

Taken as a whole, a high quality camera in good focus has been observed to result in more dramatic changes in bandwidth and packet arrival times as events occur. A low quality camera inherently creates a picture which is much less detailed and dynamic and is thus less susceptible to NTA.

### 4.1.3 Encoding Application

The largest variation for detectability of events through NTA comes with the specific encoding application. The encoding standard specifies the format of data transmitted and the type and aggressiveness of methods used to compress the video. MJPEG encoding sends images at a relatively constant frame rate. This causes most event detection to come through changes in bandwidth. The H.264 standard, which allows for variable frame rate video, can have much higher variability on the arrival time of packets. An encoding standard such as MJPEG or H.264 is only a specification for conforming to a particular protocol and does not generally give details on the specific algorithms used to implement the encoder software.

Two different implementations of the same standard may be done with widely varying effectiveness in performance and viewed output quality. Commercial codec vendors attempt to produce a fast, efficient encoder with good image quality while producing a small data stream. A goal such as low data output size can make NTA event detection easier. For example, a very aggressive compression algorithm may complete quickly for a simple image block, but take somewhat longer for data with high entropy. This aggressive algorithm results in higher variability in packet arrival times allowing NTA to make a distinction between complex and simple images being transmitted.

Many encoder settings have the potential to increase or decrease event detectability. Although the upper limit on effective image resolution and frame rate come from the video camera, the encoder makes the final determination of what resolution and frame rate will be transmitted. The compression effort and frame rate may take on many different values or may be allowed to be variable at run time. For fixed settings, this may affect the character of the video stream for its entire duration, while settings allowing for variable performance may result in more dramatic changes in NTA throughout the transmission.

Finally, the most observable, and possibly most difficult to mask, event from the encoder is when video transmission starts or stops. Due to the inherent bandwidth requirements of real time video transmission, the beginning and end of a video stream are easily observable and identifiable in NTA.



#### 4.1.4 Encryption

Of notable significance for traffic analysis of streaming video is the relatively small effect that commercial video stream encryption has on the ability to detect events. Encryption of data across networks has come as an afterthought, and the desire to minimize transmission latency and limit overhead have constrained its effectiveness in obscuring transmitted video content.

This work does not dispute the effectiveness of current encryption standards in obscuring specific image content. Unencrypted video would certainly be more vulnerable to eavesdropping than the methods currently employed. However, the detection methods and leakage sources mentioned so far are based essentially on packet arrival time and bandwidth analysis.

A specific goal of streaming encryption algorithms is to minimize the latency introduced by encryption. The time to encrypt each block of data also needs to be constant to avoid introducing a side-channel attack vector into the encryption algorithm itself. This results in a small, constant time impact to each video packet sent. Again, to avoid information leakage in encrypted output analysis, current encryption algorithms may pad data before encryption. This is generally only up to a particular block or minimum output size and these amounts are small by comparison to video transmission bandwidth [21] [22]. Since latency and bandwidth are essentially maintained through these goals, NTA techniques already mentioned are not significantly affected by application layer encryption techniques.

Other considerations include the case where only the image payload may be encrypted rather than both the payload and image headers. If image headers are left unencrypted then the data contained in those headers contains more specific and detailed information about the relationship between frames and other image blocks which would be vulnerable to attack.

Application layer encryption techniques, including AES and SSL, encrypt the network packet payload but not the packet headers themselves, which include the source and destination IP addresses as well as the source and destination ports. The port number in particular

allows the data stream for a particular application to be uniquely identified. This information has been used in this work to easily separate video traffic from other network traffic prior to packet arrival time and bandwidth measurements. Encryption provided through a VPN would somewhat obscure address and port information, but depending on the bandwidth of other traffic being transmitted along the same path, the video bandwidth may sufficiently overshadow other network traffic so as to render this extra effort ineffective.

#### **4.1.5 Computing Platform**

Expecting that variations in the complexity of image processing drives the detectability of events in network traffic, the capability of the computing platform is again expected to shape the traffic patterns. The processor speed and available system memory will set limits on the capabilities of the encoding software. The upper limits of processing speed and memory capacity can be considered fixed for the duration of a single video call, but the portion of those resources relegated to the video chat application may vary throughout the call.

We have mentioned the frame rate of the video as affecting the arrival time of network packets, but for a variable frame rate codec the image content complexity may not be the only driver for change. Other applications running concurrently on the system will affect the resources available to the encoder. If there are many applications running and the encoding process cannot be scheduled often enough, the encoder may adapt by decreasing the frame rate or encoding effort. Such variations in scheduling frequency would be seen in the network traffic analysis and would erroneously appear to indicate activity in the captured video. Similar issues could arise for other applications actively using system resources, such as a virus scan.

Other limitations in computing hardware include the interface connecting the camera to the rest of the system. Total video bandwidth would be limited by raw image bus capacity. For dedicated camera hardware, we would expect all of these computing platform factors to be negligible or disappear entirely, making the produced traffic stream more predictable.

#### 4.1.6 Network Transmission

The state of the network and the protocols used to transmit the video stream are expected to impact the detectability of events through network traffic analysis. Each network protocol introduces a different amount of transmission overhead such as whether lost packets are ignored or retransmitted and the content of packet headers. Packets arriving very late due to a request for retransmission would be grouped with other packets arriving at a similar time, skewing both timing and bandwidth measurements. Protocols which allow for some packet loss in video transmission are common and generally improve the viewing experience. For NTA of encrypted traffic, dropped packets would appear as longer delays between packets and lower bandwidth. Some video protocols use feedback from the target system to allow dynamic adjustment of encoder settings based on network conditions. As conditions change, the target bit rate of the encoded video may be increased or decreased. This would be observed in the network traffic as a change in bandwidth but is not the result of a recorded video event.

The available network bandwidth will vary dramatically between network types. A high bandwidth, low latency network is expected to leave packet timing data best intact. For wireless networks the likelihood of packets being lost, possibly requiring retransmission, is increased which would skew the measurement of timing information. Although some jitter is expected in timing measurements, the standard deviation from the mean end to end transmit time would need to be low in order to use packet timing data for event detection. Variations in end to end transmit time could result from the specific routing of packets across the network, the workload of switches or routers along that route, the medium used for transmission at each hop, or other unique conditions.

The tap point for packet collection would also have an effect on the data collected. Packets collected near the transmitter would not experience the additional jitter and timing variations introduced by network traversal, but if the total network path is clear enough those measurements may not be strongly impacted. It may be possible to perform NTA at a midpoint assuming that the route was predictable and consistent.

#### 4.1.7 Traffic Analysis

A number of factors have been considered for NTA appropriate to video stream analysis. We can use shallow packet inspection to gain information about the incoming packets. [28] The packets can be filtered by IP addresses and port number in order to isolate a specific video stream from other network traffic. For an uncongested network, the inter-arrival time of packets may signify the rate at which data is being generated by the encoder. Assuming that all data associated with a particular frame is sent as soon as that frame has been encoded, those packets will arrive in a group closer to each other in time than to packets associated with the previous or subsequent frame. This allows each frame's data to be isolated from other frames. Packet timing can therefore serve as an indication of frame rate, and variations in frame rate can be associated with changes in encode times due to captured activity. For video data transmitted from a file rather than in real time, no inference could be made based on arrival time of packets, but for real time data which is transmitted as soon as it is produced, in order to achieve low latency, packet arrival time will be indicative of encode rate.

Packet size may be an indicator of packet content. For example, in the Skype application, some similarly sized packets are sent at a regular interval even when no video is being transmitted. Packet size therefore could allow some sub-classification of packets as either video content or application overhead. By identifying and removing extra application packets before analyzing the remaining packets as video content, event detection may be improved.

Video stream bandwidth variations over time can serve as an indicator of activity since simple images are encoded at a smaller size than complex and therefore less compressible images. A significant drop in bandwidth could be an indication of changes in lighting while a smaller, but still noticeable change could indicate more localized image content changes such as a moving object.

The number of packets received in a time window is essentially a rough estimation of the average inter-arrival time of packets in that same time frame. This metric on its own yields information but is not as precise as direct measurement of packet arrival times.

A number of statistical analysis methods have been considered and employed in NTA for this work. The moving average helps normalize noise in packet measurements. Values outside some standard deviation from the moving average indicate that an event of some type has occurred, either in the recording itself or from other factors introduced throughout the video transmission pipeline. As mentioned with packet size, by detecting and separating small packets appearing at a regular interval as application overhead, the remaining packets may be more effectively analyzed. This is essentially a frequency analysis of packet size over time. Other frequency analysis may reveal additional information.

## ***4.2 Initial Investigation***

This section details the experiments we performed to determine to what extent information leakage is occurring in live video streaming, including investigation of the source of the events being measured. Analysis of the experimental results is included with the findings in this section.

### **4.2.1 Automated Test Environment**

Our initial experimentation aimed to produce the same phenomena observed in the previous VoIP work and determine both what types of events could be detected and what factors impacted the ability to detect recorded events in NTA.

An initial proof of concept by our group [29], performed analysis of network traffic recordings from Wireshark which were post processed using MatShark, a portion of the SharkTools [11] package. This method of post processing often made identifying events unreliable. Movement or actions taken in front of the camera would sometimes register as small changes in bandwidth or inter-arrival time, but other measurement noise made it difficult to determine whether analysis was being performed correctly or if false positives were being measured as successful event detection.

In order to improve the ability to tune detection algorithms and understand how various types of events appeared in the network traffic, we switched from SharkTools post processing

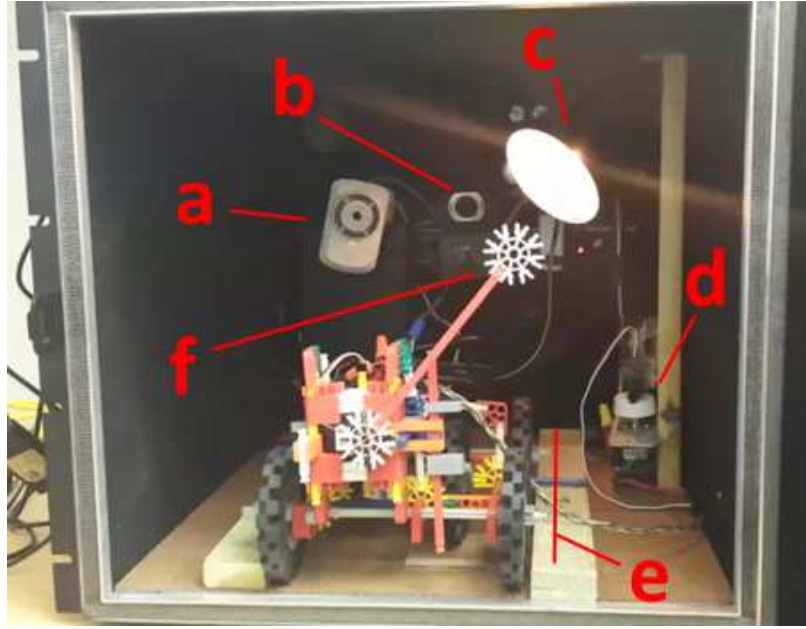


Figure 5: Light box for controlled test environment.

to real time processing using Python’s Pcap interface. Pcap essentially wraps the libpcap library making live packet capture data, equivalent to that obtainable through Wireshark, available in a programming environment. Through use of Pcap and Python’s interface to Matplotlib, real time graphs of network traffic statistics could be plotted and inspected for indication of events. This is the method used to obtain the captures shown in Figures 6 and 7.

With a framework for inspecting live packet capture data established, the next step in experimentation was to create an environment where reproducible video captures could be performed. A small radio frequency isolation chamber was re-purposed as a light box where cameras could observe a subject in a controlled environment as shown in Figure 5. Using an Arduino micro-controller board with pulse width modulating outputs, several servos could be programmatically controlled through a serial connection. After creating a simple command interface to the Arduino, single key commands could be used to control the movement of a simple vehicle and analog dimming switch.

Each component of the test environment is labeled in Figure 5: a, is a D-Link DCS-932L security camera which is configured for our tests to produce standard definition (640x480

resolution) MJPEG video; b, is a Logitech c615 web camera which captures raw high definition video at 1920x1080 resolution; c, is an incandescent lighting source; d, is an analog dimmer switch with an attached servo that allows light intensity in the test environment to be increased or decreased to arbitrary brightness; e, shows the forward and backward path of motion for the test vehicle; f, is a pivoting arm which can be rotated left or right at a controlled speed. The combination of forward and backward motion with left to right movement is designed to simulate fast and slow objects of varying size moving past the camera at varying but reproducible rates. When the chamber door is closed all light observed by the cameras in the environment is regulated. Not visible in this figure is a software controlled power switch which allows immediate switching from full brightness to no light and back as would be encountered when a light was switched on or off in a dark room.

#### **4.2.2 Event Detection Across Multiple Codecs**

Using this controlled test environment, we observed network traffic graphs as video was transmitted using each of the cameras as well as through multiple encoding applications. For the security camera, which contains its own network interface, the traffic was transmitted directly from the device to the capturing computer platform through a single hop over a local area network. For the video chat applications, a USB webcam was connected to a second computer. Both the transmit and receive computers required an Internet connection in order to log into video chat services, but both Google Hangouts and Skype established a direct connection across the local building network once the connection was negotiated.

We verified that video transmission of a still scene yielded a stable measurement of packet average inter-arrival times regardless of the camera or encoding application. Some variation from this stable level was observed at the beginning of the transmission, but after five seconds of capture average inter-arrival time (AIT) measurements would level out under any of our capture methods. This stable rate varied between video sources, but is shown in Figures 6 and 7 as approximately 7500  $\mu$ s. The MJPEG video from the D-Link camera was the most consistent in video transmission measurements with a similar measurement of

AIT and bandwidth between measurements made on separate days. The video chat services, however, seemed to negotiate a different bit rate each time a new chat was started. This is consistent with our understanding of H.264 and video chat protocols. In order to ensure the best quality of service for a variety of users, an original video connection was made at a lower bit rate in order to ensure that the call can be established. Once the connection is stable and there is no measurement of dropped packets or other factors limiting video quality, the protocol will attempt to negotiate a transition to a higher bit rate that is capable of providing a better picture to the user. Across multiple connections, we generally observed a traffic signature similar to that shown in Figure 6, but for connections originating from an outside network or for more congested local traffic conditions, the measurements would vary.

For each connection type, once a stable connection was established, the average inter-arrival time, packet sizes, and connection bandwidth varied little over time for a still scene. Once a stable connection was observed we would then initiate events by repositioning the test vehicle or changing the lighting conditions. By positioning the test vehicle close to the camera (a few inches away) and moving the arm in the field of view we could simulate a large object moving. Under good lighting conditions and with a perceived large object moving, we could observe the change in network traffic both from the Skype and Google applications. We did not see any noticeable change from the D-Link, but this is consistent with our understanding of the codecs.

For MJPEG, which does not take advantage of temporal compression, every frame is transmitted in its entirety. The images are compressed, but for an object that moves within the image where the background is uniform there is no significant change expected in the compressibility of the image. Thus, traffic measurements remain constant through the time frame of the motion.

For Skype's H.264 encoding, as shown in Figure 6, as well as for Google's VP8 encoding, no figure shown, we do observe this motion reflected in network traffic measurements. Most significantly, the average inter-arrival time of packets increases consistent with a short delay in transmission of the packets for the frame containing motion. This is again consistent



with our understanding of the H.264 and VP8 codecs which do take advantage of temporal compression techniques. We discuss this phenomenon in greater detail in Section 4.3 but, conceptually, for a still scene the encoder can transmit a delta of the previous frame which is essentially identical to itself. Thus the computation needed to encode the new information is reduced, and the quantity of information that must be sent is similarly small. When motion is added to a scene, more time is required to compute the new encoding; during this time no new information is sent. The delay is short enough not to be noticed by a human observer but stands out in measurements accurate to the microsecond level. This appears as an increase in packet inter-arrival time, a decrease in average packet size during that gap, and a corresponding drop in the closely related connection bandwidth measurement.

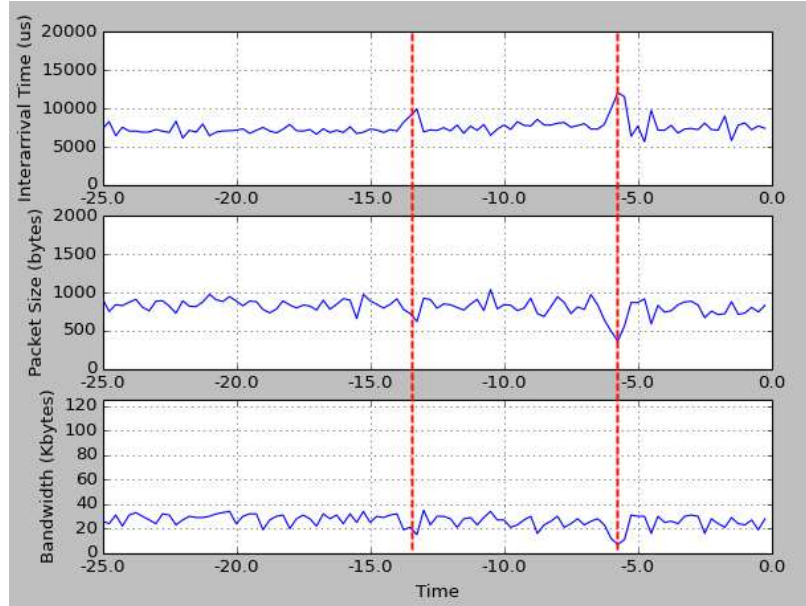


Figure 6: Skype network capture marking time of object passed in front of camera.

Variations in the size and speed of the moving object were observed in NTA but not with a linear relation to movement itself. When moving the test vehicle to the farthest position from the camera most movements of the arm were not observable. It is expected that at that distance the moving object occupied such a small portion of the image as to be negligible in comparison to other noise factors. Also, in other experiments outside the light-box a more dramatic change in the scene, which completely changed the scene content in a short time frame, would register as a large anomaly in the average inter-arrival time

(AIT) of packets. For example, this would occur each time we opened the door to the light box.

Another significant factor observed in the measurements was the quality and intensity of lighting in the scene. In a very dim scene, even a large object’s motion did not register in traffic measurements. Objects observed under low light lose color content and with low light the contrast between objects is reduced. This reduces the complexity of the scene and corresponding encoding effort. It follows that, if it is difficult to detect a moving object visually, it will be even more difficult to observe that change in the network traffic.

Correspondingly, dramatic changes in lighting were found to be very observable in network traffic and not only for the H.264 codec but also for MJPEG. Figure 7 shows a network capture from the D-Link camera’s MJPEG video which demonstrates a dramatic change in all three of the metrics. At the time indicated by the dashed red line on the left, the lights in the test environment were turned quickly off and then back on again approximately 5 seconds later.

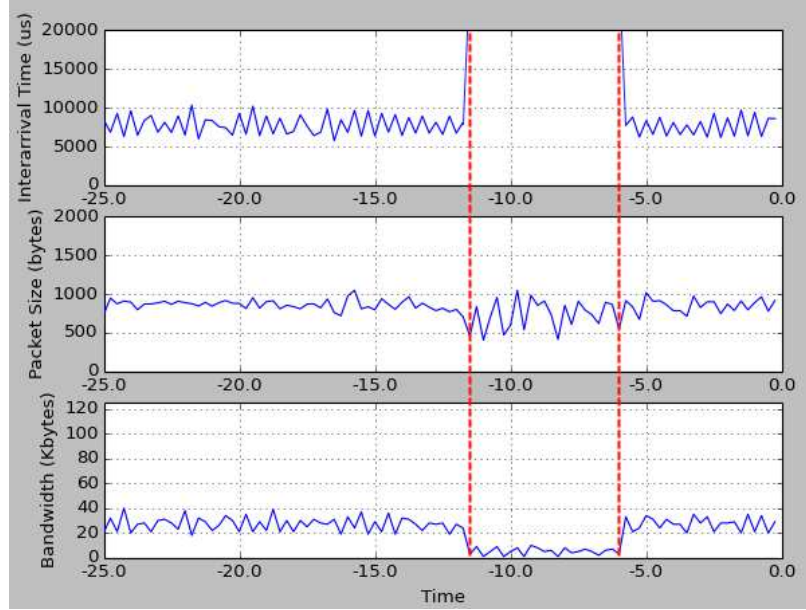


Figure 7: D-Link network capture marking time when lights are turned off then on again.

Similar reactions were observed for both the Skype and Google Hangouts recordings where changes in arrival time, packet size, and bandwidth all corresponded with the same moment that the light in the room was turned off and then back on again. Packet capture

graphs for these applications are included in Appendix B. The specific reason for the changes in network traffic between the different types of codecs is expected to be different, but the base understanding of how each of the codecs works is consistent with the network traffic analyzed. For MJPEG video the dramatic change in the complexity and resulting compressibility of the captured image, based on brightness, would change the size of the transmitted frames and appear as a change in both packet size and bandwidth. The specific reasons for changes in inter-arrival time for an MJPEG codec are still uncertain but could be a result of the codec vendor implementation that slows down traffic when there are only black frames to transmit.

For the Skype and Google applications, a change in inter-arrival time seems to be consistent with a change in frame rate. For low complexity frames with similar content, the frame rate was dramatically decreased. This is expected to be in relation to a conservation of network bandwidth where there would be almost identical frames transmitted. If this were detected, the frame rate could be significantly decreased while still presenting a good representation of recorded activity to the end user. In all three cases the decrease in complexity of the image would result in higher compressibility and decreased bandwidth utilization. This would vary based on how much change in light there was. In our test setup as we varied the light in small increments, there was not generally any dramatic change in the network traffic. If the light were only slightly dimmed in the test setup, or in a real world case if the light in a room slowly dimmed as the sun set, there would be no immediate indication of an event occurring, but the overall change would be measurable as the bandwidth transitioned from a high to a low state.

Table 2 summarizes several video applications that were tested, with the corresponding video encoding standard implemented by that application, effective methods for detecting various types of events, and by which metric they were detectable. Light intensity changes were detectable both through changes in the average inter-arrival time of packets as well as bandwidth. These indicators were less well defined in the case of Skype traffic. Small changes in light intensity were observable in all of the codecs except Skype. Dramatic changes such as lights off to on were observable in all codecs tested. Motion is detectable

Table 2: Test cases and detection methods.

Source	Codec	Detection Method				Encryption
		Small Light Changes	Lights On/Off	Small Motion	Scene Content Change	
D-Link Camera	MJPEG	AIT/BW	AIT/BW	BW	BW	None
Skype	H.264	-	AIT/BW	AIT/BW	AIT/BW	AES-256
x264	H.264	-	AIT/BW	AIT	AIT	None
Google Hangouts	VP8	AIT/BW	AIT/BW	AIT/BW	AIT/BW	SSL
WebRTC	VP8	-	AIT/BW	AIT	AIT	SSL

*AIT* - Average Inter-arrival Time

*BW* - Bandwidth

through the average inter-arrival time of packets for all but the MJPEG codec. For each codec where motion was detectable, a variable frame rate is supported and is expected to be the source of the AIT change. Even with MJPEG, a dramatic change in observed content – such as a complete background change or a large subject matter change as in a person occupying most of the frame moving out of view – would be observable, but this is more of a representation of change in image content compressibility than of motion itself. For the other codecs, smaller moments such as a hand wave or adjustment in sitting position could be observed through NTA.

For all of these observations given in Table 2 the tests were performed, as previously mentioned, over a local area network. Additional noise and complexity introduced through larger networks or over longer distances such as an Internet connection would change the detectability and decrease the ability to observe events through network traffic analysis. Some of these specific effects are discussed further in Section 4.5.

It is again noteworthy that three applications noted in Table 2 used some form of encryption, and while the content of each individual frame observed is not decipherable, correlation can still be made between the images and events transmitted and the network traffic metrics indicated.

### ***4.3 Encoder Timing Analysis of x264***

To better understand the relationship between an encoder software implementation and the resulting network traffic signature, we decided to measure the time required to encode frames compared to the resulting network packet arrival times. We chose to make our measurements for the x264 [9] open source implementation of the H.264 codec. This is the same codec standard used by the Skype application, and while the specific implementation of the standard is expected to be very different between the two software packages, the types of algorithms necessary to produce a video stream for each application are expected to be similar. This section explains how we obtained our measurements of x264 encoder execution time as well as the correlation between those measurements and the network packet arrival times.

#### **4.3.1 Algorithm Time-stamping**

The x264 application functions as a stand-alone encoder but also integrates with the GStreamer [6] framework which offers a simple command line interface to specify and then instantiate a complete video pipeline. To allow code modification, we built each piece of the GStreamer framework from version 1.2.3 of the source code. This required a rebuild of yasm-1.2.0 with no changes to the sources. We started from x264 version 0.120.2151 which we built as a Linux shared library and then linked with GStreamer framework through its plug-in interface.

Both x264 and GStreamer are written using the C language. Timing measurements were added to the `x264_encoder_encode` function in the `encoder.c` source file. Using the `gettimeofday` function available through `sys/time.h` we were able to obtain timestamps with microsecond precision. A call to `gettimeofday` was added as the first operation in the `encode` function and then again as the last operation before the function returned. These timestamps were then written to a log file on disk. It was verified that the time required to obtain the timestamps and write them to the log were negligible by comparison to the time elapsed between measurements.

We also made measurements of the time elapsed between frame encoding completion

and video payload transmission by adding an additional time-stamp log in the GStreamer udpsink plug-in. We found that there was a consistent time delay of 400 to 500  $\mu\text{s}$  between encoder completion and the call to `g_socket_send_message` regardless of events occurring in view of the camera.

The GStreamer pipeline definition used in our experiment, which is executed on the command line from a bash terminal, is given in Figure 8. The first assignment to `LD_PRELOAD` before the execution of `gst-launch-1.0` sets an environment variable which specifies that the `v4l2convert.so` library must be loaded prior to launching GStreamer and is required in order to use a video for Linux source in the pipeline. The next line launches the GStreamer application specifying verbose output and enabling debug printouts for the `v4l2src` plug-in. The video pipeline itself is then specified with each portion of the pipeline separated by an exclamation point. For the sake of clarity, and by common convention, line returns are escaped with a backslash to allow each stage of the pipeline to be placed on a separate line.

```
LD_PRELOAD="/usr/lib/i386-linux-gnu/libv4l/v4l2convert.so" \
gst-launch-1.0 -v -e --gst-debug=v4l2src:4 \
v4l2src device=/dev/video0 \
! 'video/x-raw,format=I420,width=1920,height=1080,framerate=30/1' \
! x264enc speed=preset=veryslow tune=zerolatency bitrate=800 \
! mpegtsmux \
! queue \
! chopmydata max-size=1366 \
! queue \
! udpsink host=10.0.0.1 port=5000
```

Figure 8: GStreamer pipeline construction.

We first specify that our input will come from the webcam device at `/dev/video0`. We give the format, resolution, and desired frame rate of the video. I420 here corresponds to the YUV-420 pixel format. We then specify that we will use the x264 encoder and give the settings that will be used with the encoder. We specify a target bit rate of 800 kbps with the encoder tuned to prefer low latency over high quality encoding. The encoded output is then packed for an MPEG transport stream with maximum sized payload for each packet set at 1366 bytes. Queue stages in the pipeline create FIFO buffers between stages to prevent overflow. Finally, the encoded video is transmitted via UDP on port 5000 to the destination

host.

The general construction of this pipeline is designed to transmit a high quality low latency video stream representative of modern video chat applications. The details most likely do not match any specific chat application but the traffic measurements allow us to see the relationship between real events captured by a camera, encoder performance, and network packet arrival timing.

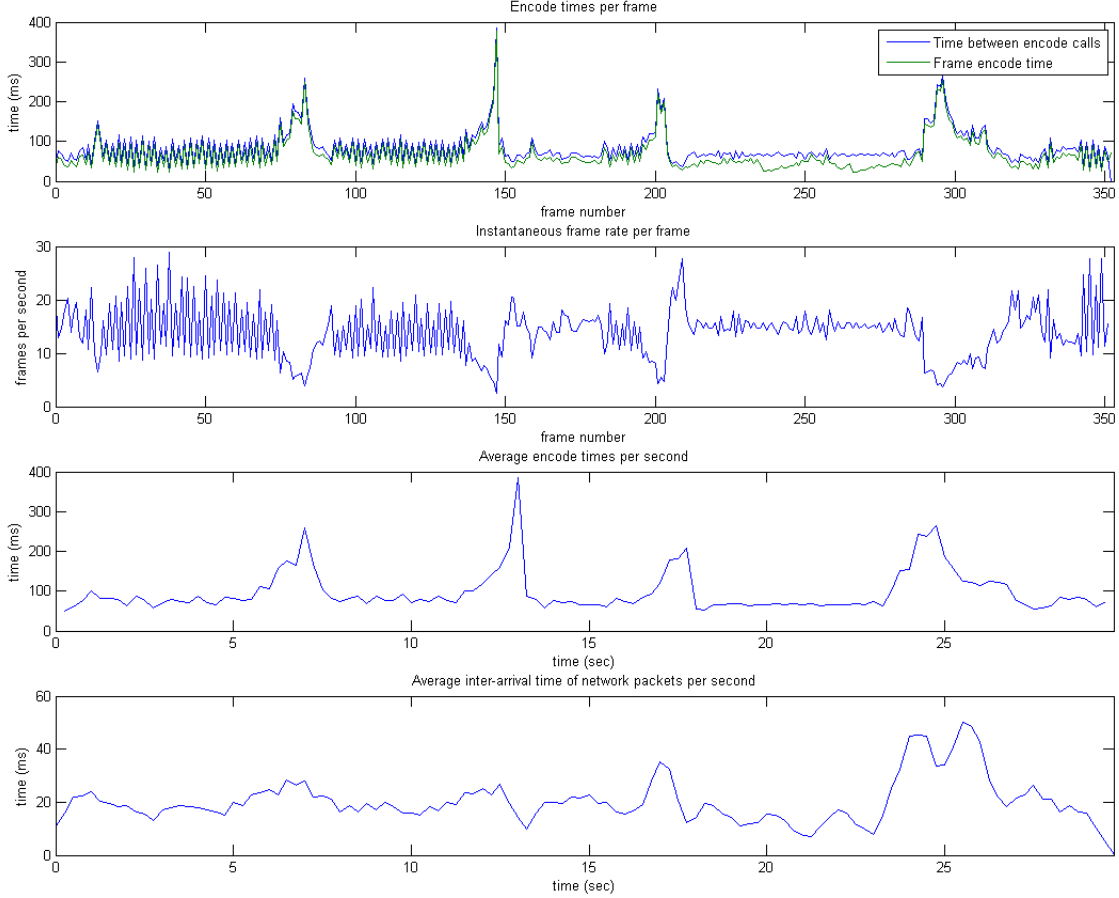


Figure 9: Open source x264 encoder performance profiling.

#### 4.3.2 Analysis of Timing Results

Figure 9 highlights the close connection between frame encode time and network packet arrival times. The topmost panel shows timing measurements of the encode function which demonstrates that time elapsed between calls to encode each frame is driven by the time required to complete the encoding of the corresponding frame. The y-axis shows time

measured in milliseconds while the x-axis indicates the frame number processed in the call to the encoder. In this test over 350 frames were encoded, each by a single call to the encode function.

The plot shows four spikes above 200 ms around frame numbers 75, 150, 200, and 300 as well as a smooth area in the measurements from frame 200 to 300. These features correspond to events occurring in view of the camera during the encoding process. While sitting in front of the camera I passed my hand through the field of view approximately two feet from the lens corresponding with frame 75. This action was intended to affect approximately 50% of pixels in the frame. The spike at frame 150 corresponds to passing my hand through the field of view approximately 2 inches from the lens. This causes the entire field of view to change as the hand enters and exits the frame. Starting at frame 200 the camera is completely covered to block any light and remains covered until approximately frame 300 where the camera is uncovered again. At other times during the test I sat motionless several feet from the camera.

We see from these results that the time required to encode a frame varies dramatically based on the motion and brightness of the scene in question. As an aid, the second panel shows the effective frames per second (fps) processed by the encoder. This is computed as the inverse of the time between encode calls from the first panel. The frame rate is fairly stable around 15 fps except when activity occurs. This is despite the target specification of 30 fps given in the pipeline description. We see here that the x264 encoder is allowing a variable frame rate which has an upper limit bounded by the processing time for each frame. For the introduction of a moving object, which adds complexity to the encode operation, the frame rate drops momentarily. While covering the camera the frame rate is able to stabilize with the decreased complexity of processing an unchanging black image.

The bottom two frames change the x-axis from frame number to a linear time scale in seconds. This is done by averaging data within a 250 ms window to create each point on the graph which is then smoothed to reduce noise. We see a correlation between the frame encode times and the network packet inter-arrival times consistent with the understanding that the encode time is related to the presence or absence of activity during the frame.



When an event occurs, it causes the encoding to complete faster or slower; packets are then transmitted immediately following encoding completion. Assuming consistent latency between packets, variations in packet arrival times correspond with events captured by the camera.

#### *4.4 Skype's H.264 Encoder*

Although previous tests had shown that we could detect events in several types of codecs, our next step involved verifying the repeatability of tests on a single application. In order to verify that similar traffic signatures resulted from similar event types, we created an application to record network traffic and a corresponding video recording of Skype calls.

We picked the Skype application due to its popularity and wide user base as well as its use of the H.264 encoding standard [15]. We do not expect our experiments with the x264 encoder to be precisely representative of Skype's H.264 encoder performance, but since they are based on the same encoding standard the comparison is closer than most other options. As a commercial application advertising AES 256-bit encryption, Skype is expected by its users to be a secure and private means of communication. By testing against the Skype application we take advantage of demonstrating the effect of strong commercial grade encryption on the ability to detect events through NTA, which we find to be minimal.

Using the Skype4Py [4] python wrapper to the Skype 3rd party API, we are able to create event handlers for various activities in the Skype application including incoming and outgoing call initiation, call waiting, call ending and other standard Skype interactions. This application, which we have named Skype Auto-Answer, executes as a background service and attaches to a running Skype session which has already been logged into on the host machine. By detecting the beginning and end of a new video call we could use Python to execute external applications. We used tcpdump to record all network traffic to a file, and FFmpeg to make a video screen capture of the Skype window. Although it would have been preferable to gain access to the decrypted raw video stream in order to better verify our NTA measurements against video packet headers, this information is not made available

by the API.

#### 4.4.1 Traffic Analysis in a Laboratory Environment

With a method available for recording the video and network traffic, for purposes of comparison between calls, we then proceeded to make calls from various locations and hardware equipment to the lab server while executing the Skype Auto-Answer application. We were able to observe under varying conditions that the network traffic and corresponding recognizability of events through NTA was affected by camera type, computer hardware, and network connection bandwidth as previously discussed in Section 4.1 (refer to Appendix B for additional details). Additionally, we observed that when these factors remained constant, the traffic measured was consistent between calls. Figure 10 shows plots for four such calls made from the same equipment. The calling computer, a Dell Latitude E6530 laptop with a high definition Logitech c615 external camera, initiated the call from within the Georgia Tech network and the same sequence of events was recorded four times. The traffic analysis for all four packet captures are combined in the figure.

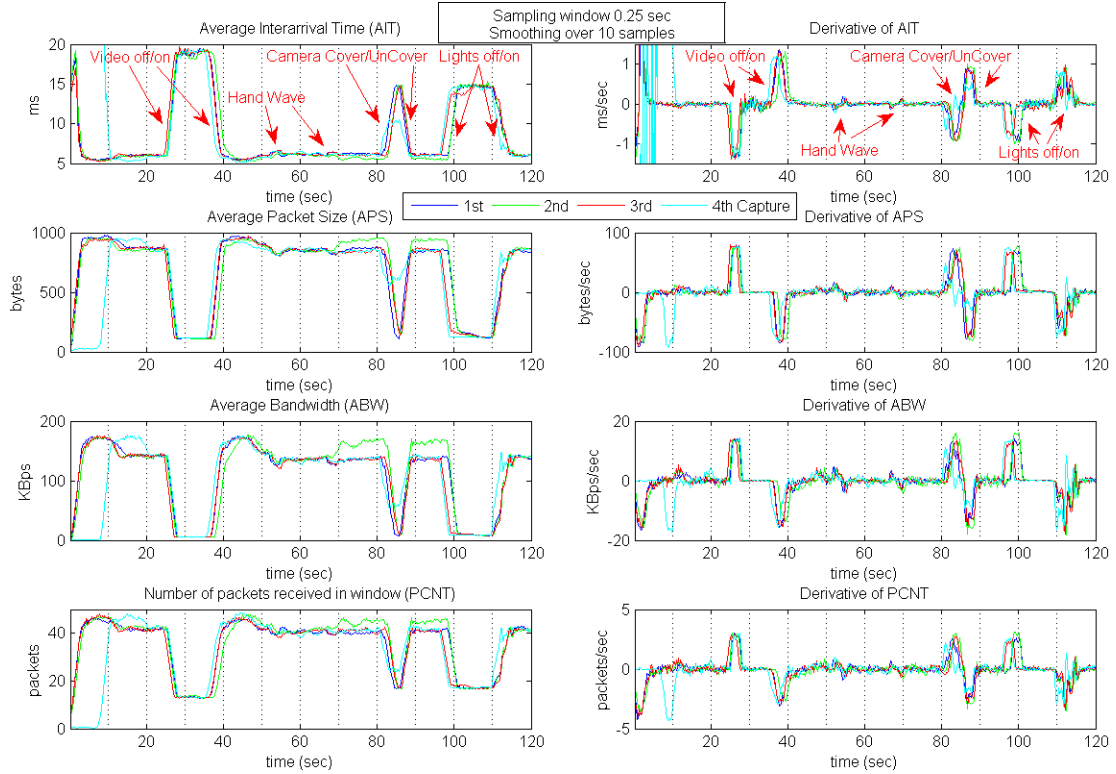


Figure 10: Laboratory environment captures showing repeatability of measurements.

The Skype call is connected and we wait approximately 25 seconds for the video stream traffic to stabilize. Generally the stream stabilizes much more quickly than this as seen in the 1st, 2nd, and 3rd captures, where stable AIT is observed after only a few seconds, but may take longer as in the 4th capture where stable traffic is not observed until 10 seconds have passed. After waiting for the stream to stabilize, we turn off the video feed in the Skype application for ten seconds then turn it back on again. Between each action throughout the call there is no movement in view of the camera. Next we move our hand past the camera approximately 12 inches from the lens, crossing through the field of view in approximately 1 second. After pausing, this action is repeated with the hand passing only 2-3 inches from the camera. Next the camera is completely covered to block out all light and remains covered for several seconds before uncovering. Finally, the lights in the room are turned off for several seconds and then back on again.

The annotations in Figure 10 show each event represented in the network traffic analysis. The eight plots show different event detection metrics. For all of these captures the sampling window is 250 ms. This sampling window is chosen based on the baseline number of packets being received. A window size that is too large will average too much data together preventing the detection of small variations such as those seen for the hand waves. A window size that is too small may not contain any packets at all. This sampling window allows us to plot the capture data on a linear time scale.

On the left hand side of Figure 10 from top to bottom, we plot measurements of network traffic metadata including average inter-arrival time (AIT) between packets, average packet size (APS), average bandwidth (ABW), and network packet count (PCNT) for packets received during the sampling window. After averaging data in each window the data set is then smoothed across 10 samples. On the right hand side of the figure is the discrete derivative of the corresponding metric from the left hand side. These discrete derivative plots are additionally smoothed over 3 samples. The derivative plots are the most useful indicators for event detection by algorithm since they are a relative measurement.

Between the four captures we observe that although there are some changes in the exact manifestation of events in the NTA, each event is consistently detectable and events

of different types are discernible from each other through careful observation. When the video is turned off we see the longest AIT since only audio and control packets are observed, whereas the camera being covered still causes some video frames to be transmitted, yielding slightly higher ABW and a corresponding drop in AIT. Covering the camera versus turning the lights on and off yield essentially the same signature with only a variation in the duration of the events. We can see, however, that as the lights are turned back on the AIT does not immediately return to the baseline level. This is due to the fact that the experiment room lighting is fluorescent rather than incandescent, and the lights turn on in a two stage process as the bulbs first warm up then reach full brightness. The hand waves yield the smallest variations and occasionally were not observable in the network traffic.

The discrete derivative plots demonstrated by these captures become more important as we attempt to perform event detection with variations in the video transmission platform. Although baseline values across these figures are very similar between captures this is not the case especially for changes in camera resolution and network bandwidth. The high and low values are indicative of activity in the video but in a more general sense the change in measurement compared to the baseline value for a specific set of hardware is a more reliable indicator.

We discovered during our original tests that we could detect events algorithmically by observing if the measurements differed from the moving average value by more than two standard deviations. Attempts to classify event types by this same method were unsuccessful, however. Through visual inspection we were able to both detect events occurring and classify those events by type in many cases.

#### **4.4.2 Event Type Classification**

To find an algorithm that would allow us to classify event types we turned to machine learning. The K-means clustering algorithm [19] allows a data set to be grouped into categories according to the distance between a particular data point and the center of each other cluster. The number of clusters is specified by the algorithm. We applied K-means auto classification to the data sets for each of the captures shown in Figure 10. Figure 11

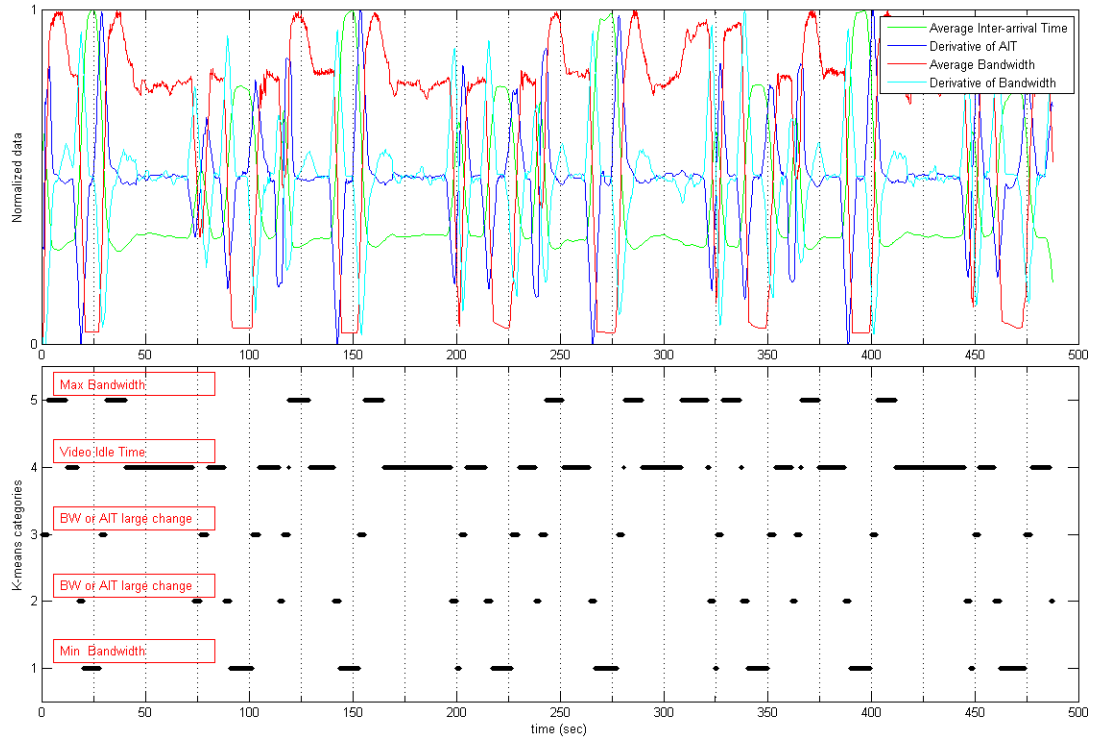


Figure 11: K-means classification of traffic analysis for laboratory video captures.

shows the resulting categorization of data points over time.

We began by concatenating the records from all four of the captures into a single data set. Timestamps were offset to make the captures appear to be a single set of data. Because of the similarities between traffic analysis metrics we limited ourselves to only two types of measurements, average inter-arrival time (AIT) and average bandwidth (ABW). We found that we obtained the best results from the k-means algorithm with smoothed data, the same as in our previous algorithm attempts. Both AIT and ABW along with their discrete derivatives were all combined as inputs for k-means. The scales of AIT and ABW are vastly different with the one originally measured in microseconds and the other in millions of bits per second. Since the k-means algorithm categorizes data based on distance to a category's mean, we first normalize the data set. This prevents k-means from artificially giving preference to either measurement since average inter-arrival times are much smaller and therefore would be inherently closer together. Derivatives are also normalized between zero and one with values less than 0.5 representing negative derivatives and values greater

than 0.5 represent positive derivatives. This approach is used rather than the absolute value of the derivatives in order to better convey the beginning and end of events.

The resulting categorization from a run of the k-means algorithm is shown in the bottom half of the figure. It can be seen that category 2 corresponds, in general, to times where the transmission was idle. Other categories correspond with various types of changes such as high bandwidth (category 5), or changing derivatives (categories 2 and 3). This does give us distinction between the beginning and end of events, but it still does not allow us to discern algorithmically between event types.

K-means auto-classification does allow us to separate idle time from periods of activity and does so with high reliability. It is possible with the k-means algorithm to train the data set for better performance by marking some of the data as belonging to a specific category. We anticipate that, if we were to specify some data points from each of the categories that we are interested in classifying, the k-means algorithm would be able to perform better than it has done in this case, possibly even allowing for the classification of events by type, but this exercise is left for future work.

## ***4.5 Collecting Outside User Video Calls***

### **4.5.1 Video Capture Collection Framework**

To understand how much variation we could expect to see between Skype calls for average users, we expanded our test setup to allow outside users to contribute to the research. We limited the contributors to laptop and desktop computing hardware, specifically disallowing smartphones and tablet devices. This had the effect of both raising the general computing ability of transmitting computers as well as restricting the network type to exclude cellular networks in most cases. While this excludes a number of users, we were still able to obtain a large set of data with a broad range of cameras and Internet connection bandwidths.

Since we intended to record video along with the network capture for each Skype session we submitted the proposed project to the Institutional Review Board (IRB). It was determined that no special restrictions would be required since the publishing of participants'

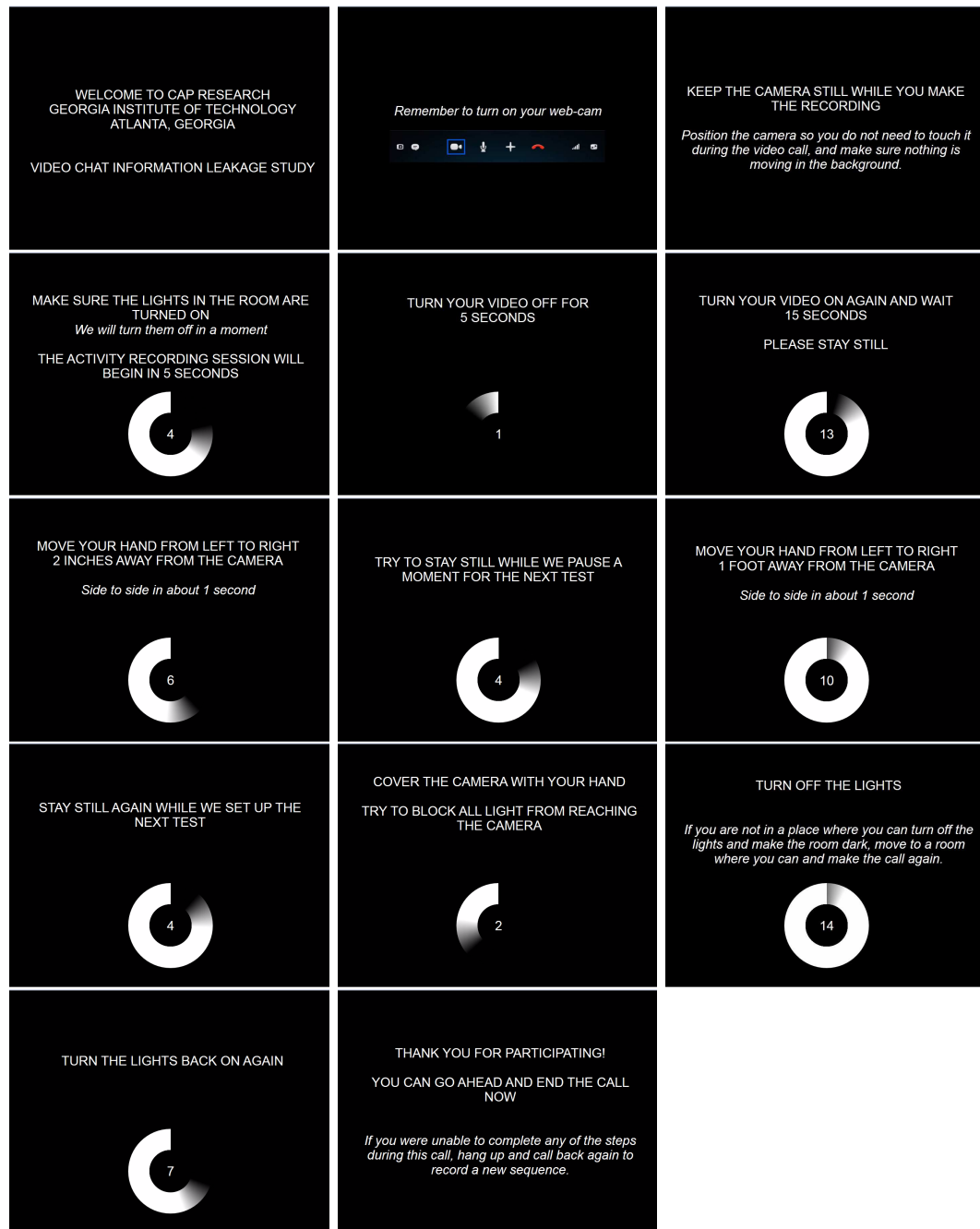


Figure 12: Excerpts from instruction video shown to participants during calls.

video content is not necessary in order to demonstrate the effectiveness of the techniques under investigation.

Again to reduce complexity in a very large test space, we decided to ask users to follow a very specific set of instructions. While recognizing that individual users would perform each task somewhat differently, we wanted to reduce that variability as much as possible in order to determine the detectability of specific types of events across computing platforms. To achieve this, a video slide-show presentation was created which could be shown to users during their session. Several frames from this video are shown in Figure 12 which is read left to right, top to bottom. The same order and timing of events from the laboratory tests were duplicated in the instruction video, including video off/on, mid-range and close up hand waves, camera cover/uncover, and lights off/on.

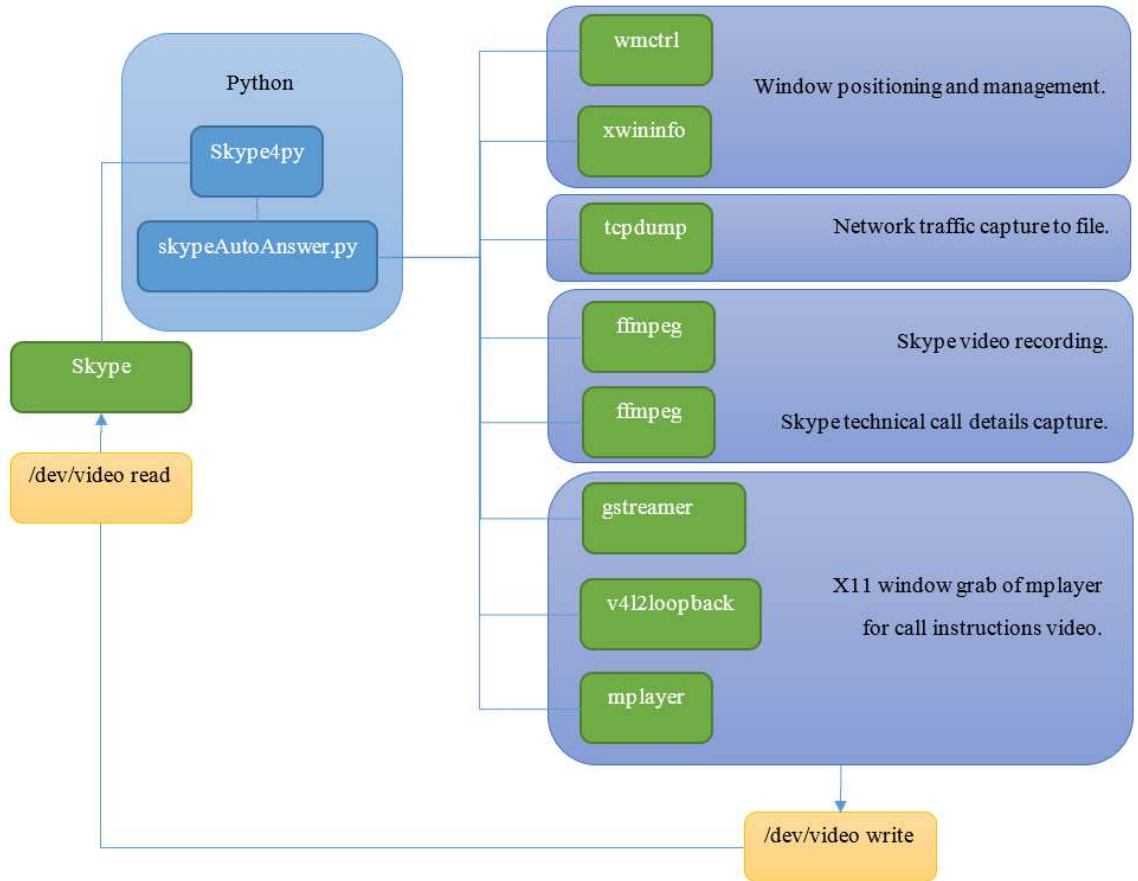


Figure 13: Control structure for Skype Auto-Answer application.

The complete control flow diagram for the Skype Auto-Answer application is given



in Figure 13, including the addition of the call instructions video playback. This was accomplished using the video for Linux loop-back module v4l2loopback [3], which creates a virtual video device in the /dev tree which can be written to in addition to read from. When the call answered event was detected through Skype4Py the call instructions video was opened through MPlayer with the playback controls hidden. The GStreamer framework was used again to grab the contents of the MPlayer window through an X11 window capture, and this input was written to the virtual video device in /dev. This same virtual video device could then be opened in Skype as the input video camera. In this way, any content playing in the MPlayer window could be seen by the participants on the far end of the connection. Code listings and scripts used for the Skype Auto-Answer application are included in Appendix A.

With the framework established for collecting video call recordings from outside users we began to recruit participants. We collected our first recordings from local university students and invited them to send a link to our project web page to others around the country who would be willing to participate in the study. We were able to collect about 30 recordings through this method, but in order to reach our goal of 100 captures we turned to Amazon Mechanical Turk [5]. Through the Amazon Turk service we were able to advertise openings in the study nationwide offering \$5.00 to each individual. The response was very quick and we were able to collect an additional 100 captures within three days of opening the study to workers on Mechanical Turk.

Prior to making their call each participant was given a brief explanation of the nature of the research and asked to fill out a survey answering questions about their geographic call location, by zip code, and details about the computer hardware setup that the call was being established from as shown in Figure 14. We collected the Skype user names of each participant in the survey which we could match up with the same profile information available through the Skype API at the beginning of each call. All of this information was saved to a database. At the end of each call another database record was created containing information about the recorded call. Many users repeated their call several times when they encountered some portion of the instructions that they were uncertain of. The database

Firefox

Video Chat Information Leakage

di-sec.org/skype\_test/hit.php

Georgia Tech

Communications Assurance and Performance Group - School of Electrical and Computer Engineering

Vision News People Publications Research Sponsors Contact

## Video Chat Information Leakage

We are investigating how effective current encryption is at hiding the content of video chat conversations. Any time an eavesdropper is able to learn anything about the conversation they are listening in on, we say that information leakage has occurred. Specifically, we are focused on an eavesdropper that tries to get information about a video call by watching how fast video packets are moving between the two computers in a conversation.

In order to help us make our measurements, we are asking you to make a short video call lasting about 2 minutes. Our research computer is set up to automatically answer your call and then record the video and network traffic during the call while you perform a series of steps in front of your webcam.

If you are willing to participate in this study and assist in furthering state of the art understanding in video information security, please begin by submitting this questionnaire.

Skype username:

The image to the right shows where to find your Skype username if you're not sure what it is. This is what you use to log into Skype not the name that appears in your contact list.  
If you do not have a Skype account go to this [link](#) to register.

Device you are making this call with: (select one)

Only calls from Desktop or Laptop computers are allowed for the study.  
Your submission is not eligible for payment with other devices.

Brand & Model (eg. Dell Inspiron, Samsung Galaxy, iPad 4)

Please be as specific as you can.

Internet download speed: (Click [here](#) to find out if you're not sure) (select one)

Webcam type: (select one)

For external webcam, list brand and model:

Again more is better than less if you're not sure.

The zipcode you are making this call from:

Have you used Skype before?

Follow the links on the next page if you need help with Skype.

If you've already filled out and submitted this form before and just need access to the call instructions please click [here](#).

Van Leer Electrical Engineering Building • 777 Atlantic Drive NW • Atlanta, GA 30332-0250 • 404.894.2901 • Fax: 404.894.4641  
© 2011, School of Electrical and Computer Engineering at the Georgia Institute of Technology • GT Legal & Privacy Information

Figure 14: Web page showing participant survey and summary of research.

Firefox

ECE Research

di-sec.org/skype\_test/hitcall.php

Google

Georgia Tech

Communications Assurance and Performance Group - School of Electrical and Computer Engineering

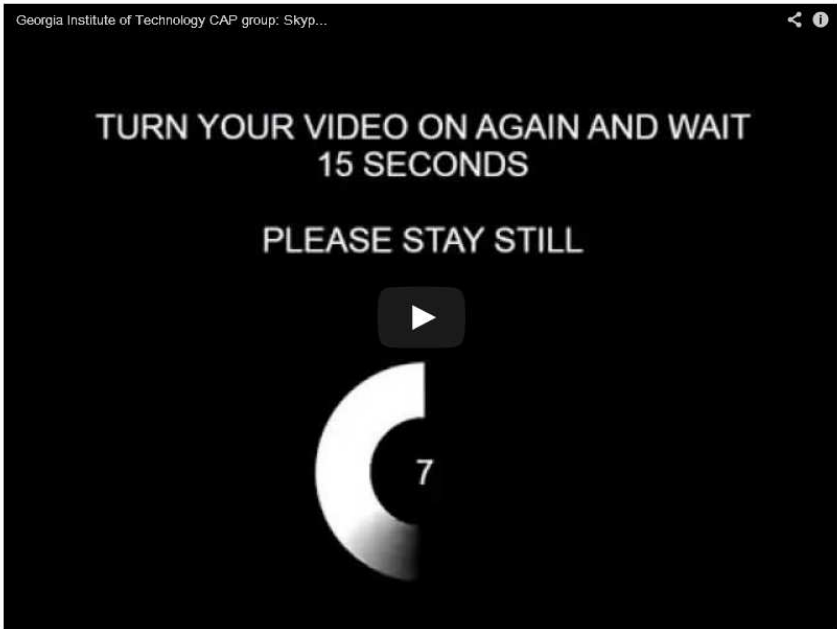
Vision News People Publications Research Sponsors Contact

## Making the Video Call

If you do not have Skype installed on your computer or need to register for a Skype username, please follow this [link](#) for installation instructions.

**Watch this video now so you know what to expect and can perform all the actions at the correct time during the call.**

This video gives step by step instructions for what you'll do after you start your Skype chat. You'll see the same video displayed on your screen during the chat session so you can perform each step at the right time.



Georgia Institute of Technology CAP group: Skyp...

During the call we'll be asking you to:

- Accept our video feed
- Turn your own video feed on and off
- Move your hand from left to right 2 inches away from the camera (side to side in about 1 second)
- Move your hand from left to right 1 foot away from the camera (side to side in about 1 second)
- Cover up your camera with your hand and uncover
- Turn the lights in the room off and on again

Figure 15: Web page showing participant instructions for conducting video calls.

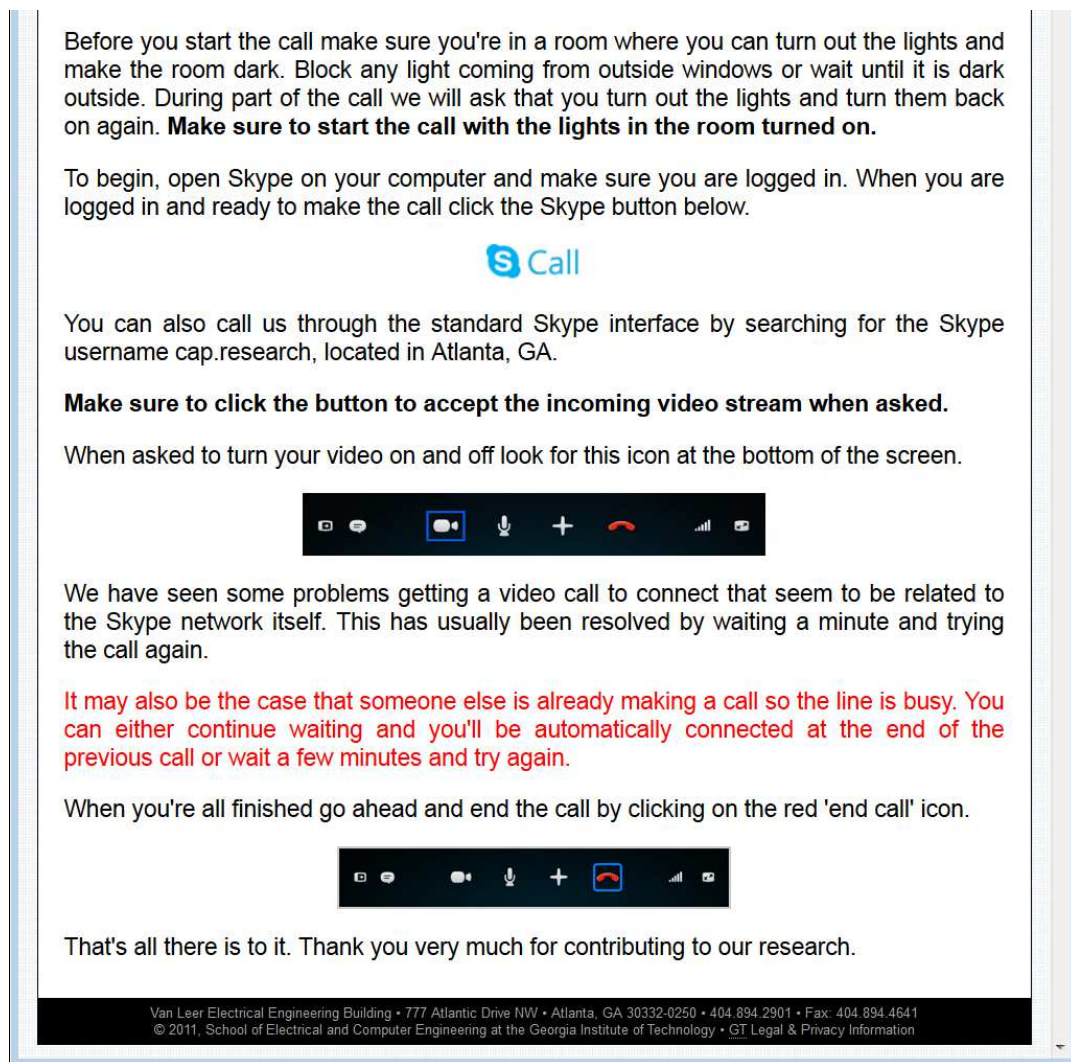


Figure 15: (continued).

records then allowed us to match up each individual user with all captures collected.

With the survey information gathered the participants were then directed to a second web page giving step-by-step written instructions about what activities would be performed during the video recording session. Screenshots from this page are shown in Figure 15. The instructional video from Figure 12 was also embedded on this page via YouTube. A link on the page was programmed to open Skype on each participant's computer and make a call directly to the research account.

As the server detected a new call ringing, it would automatically answer the call and started recording the received video, all network traffic, and technical call information



provided by Skype. At the same time the instruction video would begin providing timing cues to the user for exactly when to perform each action. A screenshot of the active Skype Auto-Answer server is shown in Figure 16. The screenshot spans a dual monitor desktop with the received Skype video filling the left monitor. The top center window show the Skype technical call information which gives details about the audio and video streams being transmitted and received as well as other application information. Below this window we see a graph of network traffic characteristics that could be reviewed in real time during the call recording. On the top right is the terminal showing the current state of the Skype Auto-Answer application. On the bottom right is the call instructions video which was transmitted to callers.

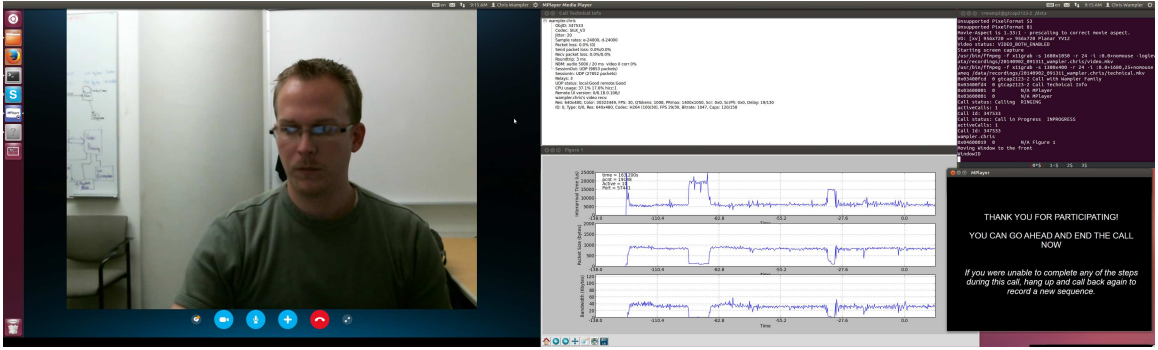


Figure 16: Server desktop view of an incoming video chat recording session.

#### 4.5.2 Overview of Collected Data from Outside Sources

We were able to capture network traffic and video from 155 unique Skype users. Each individual user may have called in multiple times in order to ensure that they were able to complete each step in the instructions correctly through the entire call. There were 287 total recordings made with 115 of these recordings verified to have been performed correctly. Of the 115 verified recordings 91 calls were established from a laptop and 24 from a desktop computer. Desktop users reported using an external webcam in 96% of cases while laptop users reported using a built in webcam in 92% of survey responses.

Figure 17 shows the self reported locations of calls originating in the United States. There are 98 calls plotted on the U.S. map with an additional 12 calls originating from India (Amazon Mechanical Turk operates in the United State and India) which are not

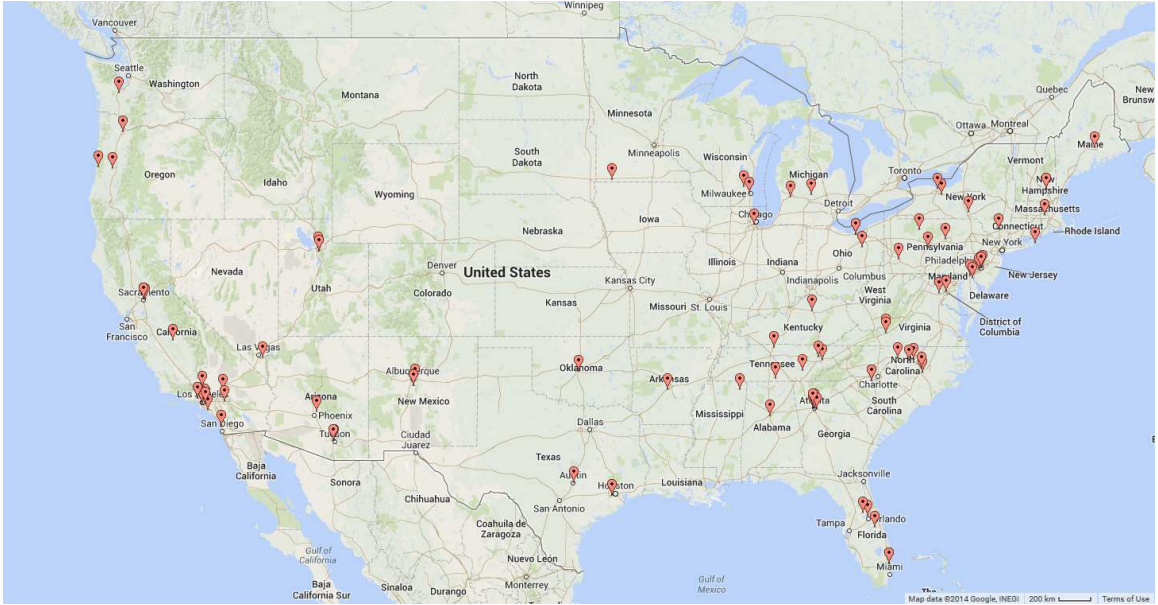


Figure 17: Locations of U.S. callers participating in experiment.

shown. Locations were established via zip code with 5 respondents declining to provide location information or providing invalid values.

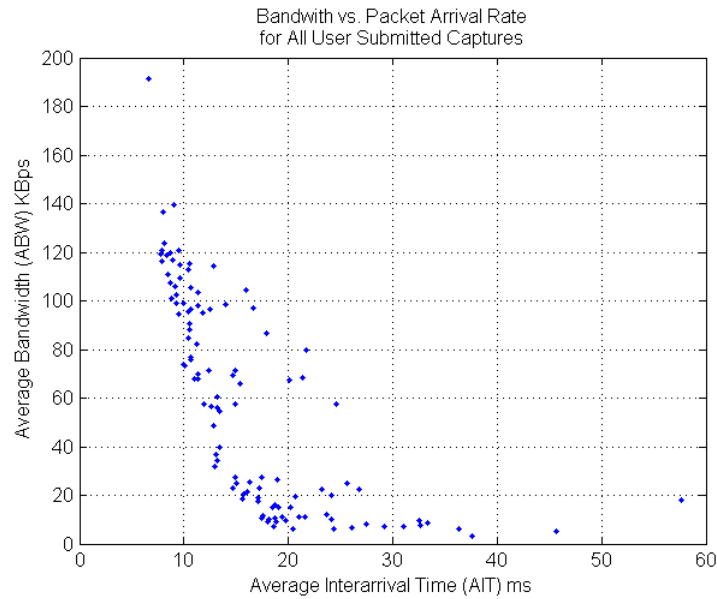


Figure 18: Variation in bandwidth versus inter-arrival time for collected data.

The variety of video stream bandwidths received is shown in Figure 18. Average packet arrival times throughout the entire capture are shown on the x-axis and average bandwidth for the captures are shown on the y-axis. The trend we observe of higher bandwidth

connections showing lower inter-arrival times is as expected. In Figures 19 and 20 we show plots of our event detection metrics from high and low video stream bandwidths respectively.

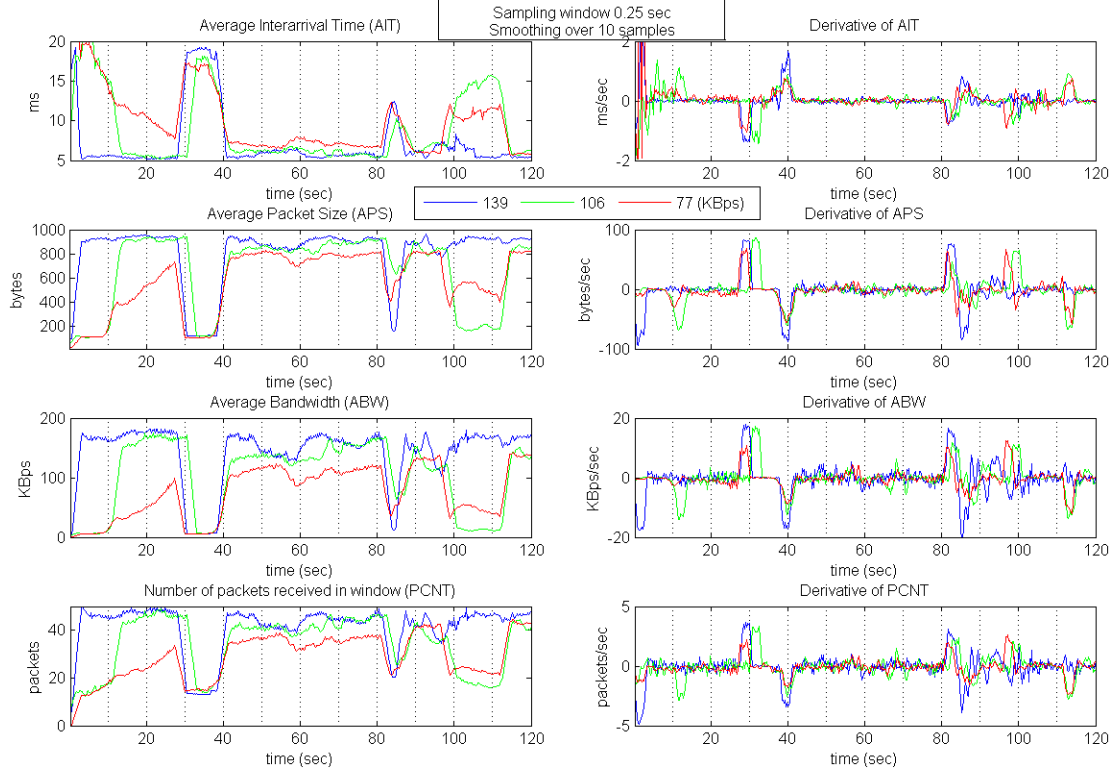


Figure 19: User captures with average bandwidths of 139, 106, and 77 KBps.

Figure 19 demonstrates several features of particular interest. At the beginning of the capture the instruction video would give a reminder to the caller to turn on their video. This reminder appeared several seconds into the video and we observe through ABW for both the 106 and 77 KBps captures that the video was not turned until 10 seconds into the recording. We are also able to see variations in lighting between the videos. At the end of the recording as participants were asked to turn off the lights in the room there was variation on how dark the room would become. In some cases the light from the computer monitor was bright enough to illuminate the room by itself. For the 139 KBps capture we recognize that the camera is covered at 85 seconds and bandwidth drops significantly, but at 100 seconds when the lights are turned off, it essentially had no effect. In this case the camera appeared to be able to gather enough light from the room to represent some detailed textures despite the main room light being turned off. This causes high bandwidth and low

inter-arrival time to be maintained and although some variation occurs, the event is not distinct. Hand waves occur at approximately 50 and 70 seconds. These small motions were difficult to discern, even at high bandwidth.

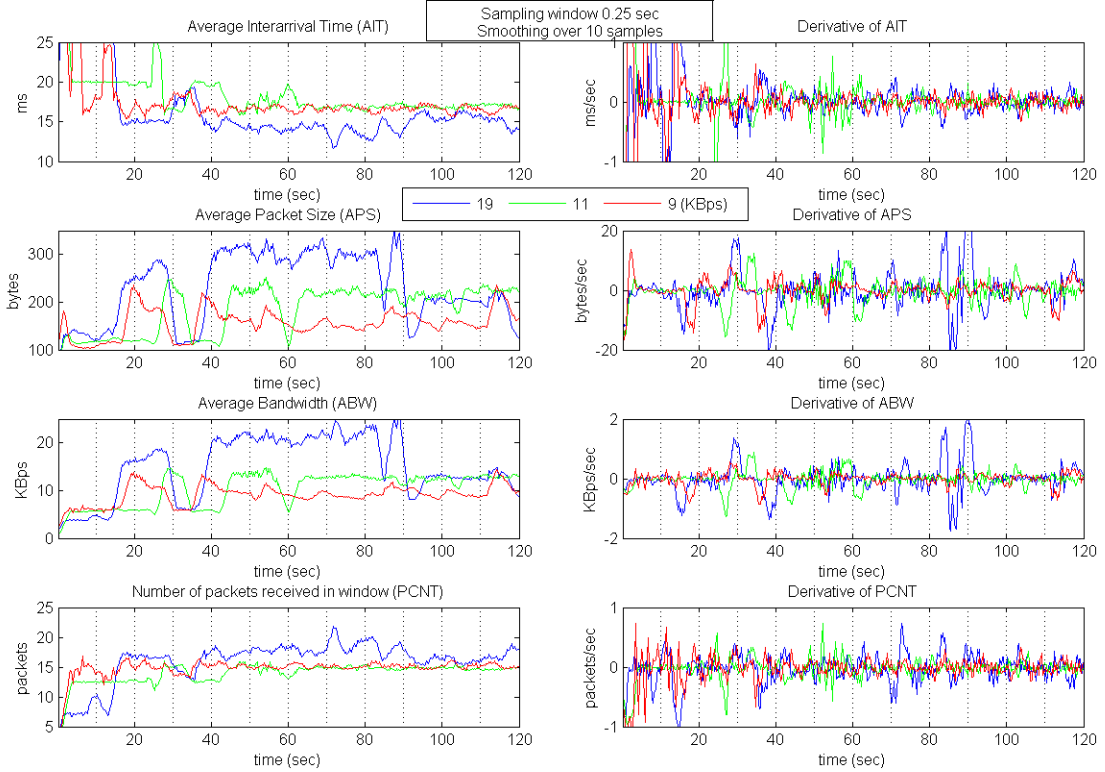


Figure 20: User captures with average bandwidths of 19, 11, and 9 KBps.

Low bandwidth captures shown in Figure 20 have had the y-axis rescaled in order to show more detail in the event detection metrics. The video being turned off is still discernible at 30 seconds in all three captures, although the participant transmitting 11 KBps was five seconds late turning off the video. Also at 60 seconds for the same user the Skype video stream stalled for a few seconds and then recovered. At these low bandwidths the only discernible event is the camera being covered at 70 seconds in the 19 KBps capture. This tendency to have events be more discernible at higher bandwidths is consistent throughout all participant recordings.



## CHAPTER V

### CONCLUSION AND FUTURE WORK

Our investigation of information leakage in encrypted video over IP traffic has found that for a variety of codecs and regardless of encryption for the tested datasets we are able to detect events occurring in the field of view of a streaming video camera through network traffic analysis. This is possible through analysis of variations in packet sizes and arrival times. We have discussed the relationship between video stream bandwidth and encoded image content. We have also measured the time required to encode frames in the x264 encoder and shown the relationship between variations in encode time resulting in similar variations in network packet arrival time.

Investigation of Skype's H.264 encoder through packet capture analysis have shown in a laboratory setting that event detection is repeatable between video calls. We have demonstrated that these events can be detected algorithmically using k-means clustering techniques. Collection of over 100 video chat sessions from outside users has shown a correlation between high network connection bandwidth and the ability to detect events.

It is anticipated that with additional effort in finding an algorithm to detect and classify events in a laboratory setting that this task can be completed. Data captured from outside sources can then be tested against the new algorithm to verify its effectiveness in detection against new video sources.

## APPENDIX A

## CODE LISTINGS FOR SKYPE AUTO-ANSWER

Code used to create the Skype Auto-Answer application is included in the listings given in this appendix. Figure 21 is included here again as a reference to the data and control flow of the application.

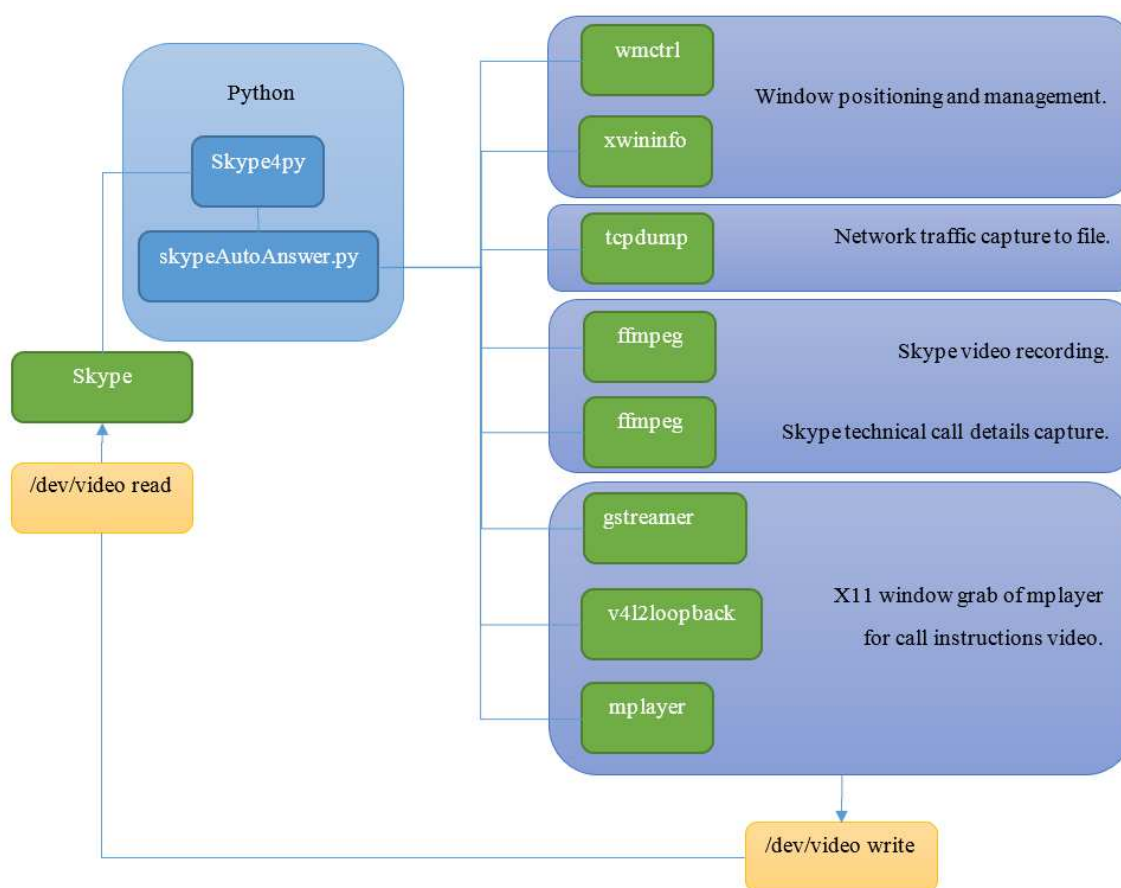


Figure 21: Control and data flow for Skype Auto-Answer application.

The script included in Figure 22 includes important steps necessary to allow the call instructions video to be played through Skype. Without the loop-back module insertion and library pre-loading when executing the Skype application, none of these features are

functional.

```
1 #!/bin/bash
2
3 # before using this script you need to first insert the v4l2 loopback module
4 # into the kernel
5 # sudo modprobe v4l2loopback
6 #
7 # also the Skype instance needs to have been started with to make it work with
8 # the video for Linux virtual loopback device
9 # env LD_PRELOAD=/usr/lib/libv4l/v4l1compat.so sudo skype
10 #
11 # the following code takes the TITLE window and uses GStreamer
12 # to make the contents of that widow act as a video device /dev/video2
13
14 # start MPlayer without console output or control through keystrokes
15 mplayer -quiet -noconsolecontrols /data/callInstructions.mp4 &
16
17 # allow some time for MPlayer to start
18 sleep 1
19
20 # get the X-Windows id for the window named MPlayer as a variable
21 TITLE="MPlayer"
22 WINDOW_XID=$(/usr/bin/xwininfo -tree -root -all | /bin/grep "$TITLE" | /bin/sed -e 's/^
    *//' | /usr/bin/cut -d\ -f1)
23
24 # put the window in the bottom left corner of the screen
25 # we need the size of the window to be exactly 640x480 for v4l2loopback
26 wmctrl -i -r $WINDOW_XID -e 0,2950,550,640,480
27
28 # launch GStreamer with an X-Windows application window as the source content
29 # and write that to the v4l2 virtual device /dev/video1
30 gst-launch-1.0 -vvv --gst-debug-level=3 ximagesrc xid=$WINDOW_XID \
31     ! videoconvert \
32     ! video/x-raw,format=YUY2,framerate=30/1 \
33     ! queue \
34     ! videoconvert \
35     ! v4l2sink device=/dev/video1
36
37 # run this command to test whether the GStreamer video device conversion is
38 # working as expected
39 #gst-launch-1.0 v4l2src device=/dev/video1 ! xvimagesink
```

Figure 22: Test script for video loop-back feature used with call instructions video.

This complete code listing for the application shows details of how the application was constructed. The code is written for Python and makes extensive use of the Skype4Py library [4].

```

1 import Skype4Py
2 import time
3 import subprocess
4 import json
5 import logging
6 import traceback
7 import signal
8 import sys
9 import urllib
10 import unicodedata
11
12 complete = 0
13
14 class Skype(object):
15     def __init__(self):#{#{
16         self.nameCounter = {}
17         currentTime = (subprocess.check_output("date +%Y%m%d_%H%M%S".split()).strip()
18         self.callId = 0
19         self.basedir = "/data/recordings/"
20         self.recordsdir = self.basedir + "Backup_Records_Files/"
21
22         logging.basicConfig(filename=self.recordsdir+currentTime + "_autoanswer.log",level=logging.
23             INFO,format='%(asctime)s **%(name)s %(levelname)s %(message)s', datefmt='%Y/%m
24             /%d %H:%M:%S')
25         self.rootlogger = logging.getLogger()
26         self.rootlogger.addFilter(logging.Filter(name='root'))
27         self.rootlogger.setLevel(logging.DEBUG)
28         self.printlogger = logging.StreamHandler(sys.stdout)
29         self.printlogger.setLevel(logging.INFO)
30         self.printlogger.setFormatter(logging.Formatter('%(message)s'))
31         self.printlogger.addFilter(logging.Filter(name='root'))
32         self.rootlogger.addHandler(self.printlogger)
33
34         self.callStart = 0
35         self.callEnd = 0
36         self.videoStart = 0
37         self.videoEnd = 0
38         self.TCP_DUMP_PROCESS = 0
39         self.VIDEO_RECORD_PROCESS = 0
40         self.TECH_RECORD_PROCESS = 0
41         self.MPLAYER_PROCESS = 0
42         self.GST_PROCESS = 0
43         self.pcap_proc = 0
44
45         self.CallStatus = 0
46         self.CallIsFinished = set ([Skype4Py.clsFailed, Skype4Py.clsFinished, Skype4Py.clsMissed,
47             Skype4Py.clsRefused, Skype4Py.clsBusy, Skype4Py.clsCancelled]);
48         self.CallIncoming = set([Skype4Py.cltIncomingPSTN, Skype4Py.cltIncomingP2P]);#}}}}
49
50     def launch(self):
51         #global nameCounter
52         ret=0
53         try:
54             self.skype = Skype4Py.Skype()

```

```

52     self.skype.OnAttachmentStatus = self.OnAttachmentStatus
53     self.skype.OnCallStatus = self.OnCallStatus
54     self.skype.OnCallInputStatusChanged = self.OnCallInputStatusChanged
55     self.skype.OnCallSeenStatusChanged = self.OnCallSeenStatusChanged
56     self.skype.OnCallVideoReceiveStatusChanged = self.OnCallVideoReceive
57     self.skype.OnConnectionStatus = self.OnConnectionStatus
58     self.skype.OnCallVideoSendStatusChanged = self.OnCallVideoSend
59     self.skype.OnCallVideoStatusChanged = self.OnCallVideo
60
61     # Starting Skype if it's not running already..
62     if not self.skype.Client.IsRunning:
63         logging.info("Skype was not running at launch")
64         self.SkypeStartup()
65
66     # Attatching to Skype – blocks with a timeout of 10 seconds..
67     logging.info('Connecting to Skype..')
68     #self.skype.Attach(Wait=True)
69     self.skype.Attach()
70     logging.info('Attach completed')
71
72     #Changing user status seems to be causing an exception
73     #making it retry up to three times before giving up
74     retry=3
75     while retry > 0:
76         try:
77             self.skype.ChangeUserStatus(Skype4Py.cusOnline)
78             break
79         except:
80             logging.error("Retrying Change User Status: " + str(retry) )
81             retry = retry - 1
82     logging.info('User status set to ' + self.skype.Convert.UserStatusToText(Skype4Py.
83         cusOnline))
84
85     # Directory for the Video Call's usernames
86     self.nameCounter = {}
87     logging.info("Loading the Record...")
88     CounterFile = open(self.recordsdir + "SkypeRecords.txt", "r")
89     contents = CounterFile.read()
90     self.nameCounter = json.loads(contents)
91
92     Backup = open(self.recordsdir + "SkypeBackup.txt", "w")
93     Backup.write(contents)
94     Backup.close()
95
96     CounterFile.close()
97
98     except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError), e:
99         logging.error(traceback.format_exc())
100         logging.error("[LAUNCH ERROR]: " + str(e))
101
102     def SkypeStartup(self):
103         logging.info('Starting Skype..')
104         self.skype.Client.Start()
105         #time.sleep(5)

```

```

105
106 def OnCallInputStatusChanged(self, status):#{#{
107     logging.info("OnCallInputStatusChanged status: " + status)
108
109 def OnCallSeenStatusChanged(self, call, status):
110     logging.info("OnCallSeenStatusChanged status: " + str(status) + " call.Id: " + str(call.Id))
111
112 def OnConnectionStatus(self, status):
113     logging.info("OnConnectionStatus status: " + status)
114
115 def AttachmentStatusText(self, status):
116     return self.skype.Convert.AttachmentStatusToText(status)
117
118 def CallStatusText(self, status):
119     return self.skype.Convert.CallStatusToText(status)
120
121 def OnCallVideoSend(self, call, status):
122     logging.info('Video Send status: ' + status)
123     if status == Skype4Py.cvsSendEnabled:
124         self.screenCapture(self.directory)
125         self.maximizeScreen(call)
126
127 def OnCallVideoReceive(self, call, status):
128     logging.info('Video Receive status: ' + status)
129     if status == Skype4Py.cvsReceiveEnabled:
130         self.screenCapture(self.directory)
131         self.maximizeScreen(call)
132
133 def OnCallVideo(self, call, status):
134     logging.info('Video status: ' + status)
135     if status == Skype4Py.cvsBothEnabled:
136         self.screenCapture(self.directory)
137         self.maximizeScreen(call)
138
139 def hideSkype(self):
140     self.skype.Client.WindowState = Skype4Py.wndHidden;
141
142 def showSkype(self):
143     self.skype.Client.WindowState = Skype4Py.wndMaximized;#}}}
144
145 def OnCallStatus(self, call, status):
146     try:
147         logging.info('Call status: ' + self.CallStatusText(status) + " " + status)
148         logging.info("activeCalls: " + str(self.skype.ActiveCalls.Count))
149         logging.info("Call Id: " + str(call.Id))
150         #placeCall(call.PartnerHandle)
151
152         if status == Skype4Py.clsRinging and ( call.Type in self.CallIncoming):
153             # Only answer one call at a time
154             if self.callId == 0 :
155                 self.startNewCall(call)
156
157         elif status == Skype4Py.clsInProgress:
158             logging.info(call.PartnerHandle)

```

```

159
160         try:
161             call.StartVideoReceive()
162
163
164
165         except Skype4Py.SkypeAPIError, e:
166             print "NO VIDEO SkypeAPIError"
167         except Skype4Py.SkypeError, e:
168             print "NO VIDEO SkypeError"
169             time.sleep(1)
170
171     elif (status == Skype4Py.clsFinished) or (status == Skype4Py.clsUnplaced): #in self.
172         CallIsFinished:
173         self.cleanUpCall(call)
174
175     else:
176         logging.warning("Unhandled status: " + status)
177
178     except Skype4Py.SkypeAPIError, e:
179         logging.error(traceback.format_exc())
180         logging.error("[ON CALL STATUS ERROR SkypeAPIError]: " + str(e))
181     except Skype4Py.SkypeError, e:
182         logging.error(traceback.format_exc())
183         logging.error("[ON CALL STATUS ERROR SkypeError]: " + str(e))
184
185 def startNewCall(self, call):
186     self.callId = call.Id
187
188     time.sleep(1.0)
189     #as soon as we have a call answered make a folder and start recording network traffic
190     self.callStart = time.time()
191     currentTime = (subprocess.check_output("date +%Y%m%d_%H%M%S".split()).strip()
192     self.calddbtime = (subprocess.check_output(['date', '+%Y-%m-%d %H:%M:%S']).strip()
193     self.directory = "/data/recordings/" + currentTime + "_" + str(call.PartnerHandle)
194
195     # Directory of each Callee
196     logging.debug("Creating directory: " + self.directory)
197     subprocess.call(["sudo mkdir " + self.directory], shell = True)
198     subprocess.call(["sudo chmod 2777 " + self.directory], shell = True)
199     self.packetCapture(self.directory)
200
201     self.counterFile(call)
202     self.answerCall(call)
203     self.instructionsCall(self.directory)
204
205 def OnAttachmentStatus(self, status):
206     try:
207         logging.info('API attachment status: ' + self.AttachmentStatusText(status) + "(" + str(
208             status) + ")")
209
210     #restart skype if it crashes or gets closed
211     if not self.skype.Client.IsRunning:

```

```

211         self.SkypeStartup()
212
213     self.skype.Attach()
214
215     except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError), e:
216         logging.error(traceback.format_exc())
217         logging.error("[ON ATTACH FUNCTION ERROR]: " + str(e))
218
219     def placeCall(self, who):
220         try:
221             logging.info('Calling ' + who + '...')
222             self.skype.PlaceCall(who)
223         except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError), e:
224             logging.error(traceback.format_exc())
225             logging.error("[PLACECALL ERROR]: " + str(e))
226             logging.error("to whom: " + str(who))
227             logging.error("skype instance: " + str(self.skype))
228
229     def cleanUpCall(self, call):
230         try:
231             self.callEnd = time.time()
232             try:
233                 user = unicode(str(call.PartnerHandle), 'utf-8')
234                 username = unicodedata.normalize('NFD', user).encode('ascii', 'ignore')
235                 username = urllib.quote_plus(username)
236             except:
237                 username = "UsernameError"
238
239             try:
240                 profile = unicode(str(call.PartnerDisplayName), 'utf-8')
241                 profilename = unicodedata.normalize('NFD', profile).encode('ascii', 'ignore')
242                 profilename = urllib.quote_plus(profilename)
243             except:
244                 print "ERROR TRACE: ", traceback.format_exc()
245                 profilename = "ProfilenameError"
246
247             try:
248                 self.calldbtime = urllib.quote_plus(self.calldbtime)
249             except:
250                 print "ERROR TRACE: ", traceback.format_exc()
251
252             curlCommand = "curl -sS 'http://di-sec.org/skype_test/computerserver.php?' + "
253                 skypeUsername=" + username + "&clientName=" + profilename + "&calltime=" +
254                 self.calldbtime + "&callduration=" + str(round(self.callEnd - self.callStart, 2)) + ""
255
256             print "URL DATABASE: ", curlCommand
257
258             subprocess.call([curlCommand], shell = True)
259
260             logging.info("endCall status: " + call.Status)
261
262             skypeCallWindowName = "Call with " + call.PartnerDisplayName
263             logging.info("closing window: " + skypeCallWindowName)

```



```

263 subprocess.call(['wmctrl -c ' + skypeCallWindowName + ''], shell = True)
264
265 techInfoWindowName = "Call Technical Info"
266 logging.info("closing window: " + techInfoWindowName)
267 subprocess.call(['wmctrl -c ' + techInfoWindowName + ''], shell = True)
268
269 if self.checkProcessRunning("tcpdump") == 0:
270     logging.info("Ending tcpdump")
271     self.TCP_DUMP_PROCESS.terminate()
272     ret = self.TCP_DUMP_PROCESS.wait()
273     logging.debug("process ended with: " + str(ret))
274 if self.checkProcessRunning("ffmpeg") == 0:
275     logging.info("Ending video recording")
276     self.VIDEO_RECORD_PROCESS.terminate()
277     ret = self.VIDEO_RECORD_PROCESS.wait()
278     logging.debug("process ended with: " + str(ret))
279 if self.checkProcessRunning("ffmpeg") == 0:
280     logging.info("Ending technical info recording")
281     self.TECH_RECORD_PROCESS.terminate()
282     ret = self.TECH_RECORD_PROCESS.wait()
283     logging.debug("process ended with: " + str(ret))
284 #end gst before closing window so it doesn't lose it's target
285 if self.checkProcessRunning("gst-launch-1.0") == 0:
286     logging.info("Ending GST")
287     self.GST_PROCESS.terminate()
288     ret = self.GST_PROCESS.wait()
289     logging.info("process ended with: " + str(ret))
290     self.GST_PROCESS = 0
291
292 self.waitForProcessEnd("gst-launch-1.0")
293 logging.info("GST shows ended")
294
295 if self.checkProcessRunning("mplayer") == 0:
296     logging.info("Ending Mplayer")
297     self.MPLAYER_PROCESS.terminate()
298     ret = self.MPLAYER_PROCESS.wait()
299     logging.info("process ended with: " + str(ret))
300     self.MPLAYER_PROCESS = 0
301
302
303 # Close the window when screenshot is taken.
304 subprocess.call(['wmctrl -c "Figure 1"'], shell = True)
305 self.waitForScreenClosed("Figure 1")
306 self.pcap_proc.terminate()
307
308 logging.info("waiting for skype window to close")
309 self.waitForScreenClosed(skypeCallWindowName)
310 logging.info("waiting for skype tech info window to close")
311 self.waitForScreenClosed(techInfoWindowName)
312
313 logging.info("Processes still running BEGIN")
314 self.checkProcessRunning("tcpdump")
315 self.checkProcessRunning("ffmpeg")
316 self.checkProcessRunning("gst-launch-1.0")

```

```

317     self.checkProcessRunning("mplayer")
318     logging.info("Processes still running END")
319
320     self.callId = 0
321     for curr in self.skype.ActiveCalls:
322         if (curr.Status == Skype4Py.clsRinging) and (self.callId == 0) :
323             self.startNewCall(curr)
324
325     except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError), e:
326         logging.error(traceback.format_exc())
327         logging.error("[END CALL FUNCTION ERROR]: " + str(e))
328     except IndexError, e:
329         logging.error(traceback.format_exc())
330         logging.error("IndexError on endCall: " + str(e))
331
332     def answerCall(self, call):
333         try:
334             call.Answer()
335             logging.info(call.Datetime)
336         except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError), e:
337             logging.error(traceback.format_exc())
338             logging.error("[ANSWER CALL FUNCTION ERROR]: " + str(e))
339         except IndexError, e:
340             logging.error(traceback.format_exc())
341             logging.error("IndexError on answerCall: " + str(e))
342
343     def packetCapture(self, directory):
344         try:
345             logfile = open(directory + "/tcpdump.log", 'w')
346             logging.info("Starting Network Capture")
347             interface = "eth1"
348             TCP_DUMP_COMMAND = "tcpdump -j adapter_unsynced -s 65535 -w " + directory
349                 + "/capture.pcap -i " + interface
350             logging.info(TCP_DUMP_COMMAND)
351             self.TCP_DUMP_PROCESS = subprocess.Popen(TCP_DUMP_COMMAND.split(), shell =
352                 False, stdout=logfile, stderr=logfile)
353         except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError, subprocess.CalledProcessError), e:
354             logging.error(traceback.format_exc())
355             logging.error("[PACKET CAPTURE FUNCTION ERROR]: " + str(e))
356
357     def screenCapture(self, directory):
358         try:
359             logfile = open(directory + "/ffmpeg.log", 'w')
360             windowsize = "1680x1050"
361             logging.info("Starting screen capture")
362             VIDEO_RECORD_COMMAND = "/usr/bin/ffmpeg -f x11grab -s " + windowsize + " -
363                 r 24 -i :0.0+nomouse -loglevel debug -sameq " + directory + "/video.mkv"
364             logging.info(VIDEO_RECORD_COMMAND)
365             self.VIDEO_RECORD_PROCESS = subprocess.Popen(VIDEO_RECORD_COMMAND.
366                 split(), shell = False, stdout=logfile, stderr=logfile)
367
368         # Running a python program to run the graph for the PCAP files.
369         pcap_command = "sudo python /data/livecapplot.py -p 57441 -i eth1"
370         self.pcap_proc = subprocess.Popen(pcap_command.split(), shell = False)

```

```

367
368     logfile2 = open(directory + "/ffmpeg-technical.log", 'w')
369     window_size = "1300x400"
370     TECH.RECORD.COMMAND = "/usr/bin/ffmpeg -f x11grab -s " + window_size + " -r
        24 -i :0.0+1680,25+nomouse -loglevel debug -sameq " + directory + "/technical.
        mkv"
371     logging.info(TECH.RECORD.COMMAND)
372     self.TECH.RECORD.PROCESS = subprocess.Popen(TECH.RECORD.COMMAND.split
        (), shell = False, stdout=logfile2, stderr=logfile2)
373     except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError, subprocess.CalledProcessError), e:
374         logging.error(traceback.format_exc())
375         logging.error("[SCREEN CAPTURE FUNCTION ERROR]: " + str(e))
376
377     def wait_for_screen(self, screen_name, retry=25):
378         while retry > 0:
379             ret = subprocess.call(['wmctrl -l | grep "' + screen_name + '"'], shell = True)
380             if ret == 0:
381                 break
382             time.sleep(.250)
383             retry = retry - 1
384
385     def wait_for_screen_closed(self, screen_name, retry=25):
386         while retry > 0:
387             ret = subprocess.call(['wmctrl -l | grep "' + screen_name + '"'], shell = True)
388             if ret == 1:
389                 break
390             time.sleep(.250)
391             retry = retry - 1
392
393     def check_process_running(self, process_name):
394         ret = subprocess.call(['ps aux | grep -v "grep" | grep "' + process_name + '"'], shell = True)
395         return ret
396
397     def wait_for_process(self, process_name, retry=8):
398         while retry > 0:
399             ret = subprocess.call(['ps aux | grep -v "grep" | grep "' + process_name + '"'], shell =
                True)
400             if ret == 0:
401                 break
402             time.sleep(.250)
403             retry = retry - 1
404
405     def wait_for_process_end(self, process_name, retry=25):
406         while retry > 0:
407             ret = subprocess.call(['ps aux | grep -v "grep" | grep "' + process_name + '"'], shell =
                True)
408             if ret == 1:
409                 return
410             time.sleep(.250)
411             logging.info("Process not ended " + process_name + " ret: " + str(ret))
412             retry = retry - 1
413             logging.error("Process did not end correctly, retry timeout")
414
415

```

```

416 def maximizeScreen(self, call):
417     try:
418         self.waitForScreen("Call with " + call.PartnerDisplayName)
419         subprocess.call(['wmctrl -r "Call with ' + call.PartnerDisplayName + '" -b add,fullscreen'
420             ], shell = True)
421         self.waitForScreen("Call Technical Info")
422         subprocess.call(['wmctrl -r "Call Technical Info" -e 0,1680,0,1300,400'], shell = True)
423         subprocess.call(['wmctrl -a "Call Technical Info"'], shell = True)
424
425         self.waitForScreen("Figure 1")
426
427         # Set window in position and resize it.
428         windows_repos = subprocess.Popen(['wmctrl -r "Figure 1" -e 0,1680,480,1370,600'], shell
429             = True)
430         windows_repos.wait()
431
432         print "Moving Window to the front"
433         windows_inFront = subprocess.Popen(['wmctrl -a "Figure 1"'], shell = True)
434         windows_inFront.wait()
435
436         WINDOW_XID = (subprocess.check_output("xwininfo -root -tree | grep skype | head -n
437             1 | awk '{print $1}'", shell = True)).strip()
438         WINDOW_XID = (subprocess.check_output("xwininfo -root -tree | grep skype | grep 307
439             | awk '{print $1}'", shell = True)).strip()
440         print "WindowID" + WINDOW_XID
441         subprocess.call(['wmctrl -i -r ' + WINDOW_XID + ' -e 0,3276,0,307,119'], shell = True)
442
443     except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError, subprocess.CalledProcessError), e:
444         logging.error(traceback.format_exc())
445         logging.error("[MAXIMIZE SCREEN FUNCTION ERROR]: " + str(e))
446
447 def counterFile(self, call):
448     #global nameCounter
449
450     if call.PartnerHandle in self.nameCounter:
451         self.nameCounter[call.PartnerHandle] = self.nameCounter[call.PartnerHandle] + 1
452     else:
453         self.nameCounter[call.PartnerHandle] = 0
454
455     # Directory for the Video Call's names
456     CounterFile = open("/data/recordings/Backup_Records_Files/SkypeRecords.txt", "w")
457     CounterFile.write(json.dumps(self.nameCounter))
458     CounterFile.close()
459
460 def instructionsCall(self, directory):
461     try:
462         logfile = open(directory + "/callInstructions.log", 'w')
463         MPLAYER_COMMAND = "mplayer -quiet -noconsolecontrols /data/callInstructions.

```

```

464     MPLAYER_Windows = 'MPlayer'
465     self.waitForScreen(MPLAYER_Windows)
466     WINDOW_XID = (subprocess.check_output("/usr/bin/xwininfo -tree -root -all | /bin/
        grep '" + MPLAYER_Windows + "' | /bin/sed -e 's/^ */' | /usr/bin/cut -d\ -f1",
        shell = True)).strip()
467
468     subprocess.call("wmctrl -i -r " + WINDOW_XID + " -e 0,2950,550,640,480", shell =
        True)
469     self.waitForScreen(MPLAYER_Windows)
470     time.sleep(.250)
471     gst_command = "gst-launch-1.0 -vvv --gst-debug-level=3 ximagesrc xid=" +
        WINDOW_XID + " show-pointer=false ! videoconvert ! video/x-raw,format=YUY2,
        framerate=30/1 ! queue ! videoconvert ! v4l2sink device=/dev/video1"
472     self.GST_PROCESS = subprocess.Popen(gst_command.split(), shell = False, stdout=logfile,
        stderr=logfile)
473
474     except (Skype4Py.SkypeAPIError, Skype4Py.SkypeError, subprocess.CalledProcessError), e:
475         logging.error(traceback.format_exc())
476         logging.error("[INSTRUCTION CALL FUNCTION ERROR]: " +str(e))
477
478     def handler(signum, frame):
479         global complete
480         global rec
481
482         #rec.skype.Client.Shutdown()
483         complete=1
484
485         #exit()
486
487     try:
488
489         if __name__ == "__main__":
490             global rec
491             signal.signal(signal.SIGINT, handler)
492             rec = Skype()
493             rec.launch()
494
495         while not complete:
496             time.sleep(1)
497
498     except Exception, err:
499         logging.error(traceback.format_exc())

```

## APPENDIX B

### ADDITIONAL TRAFFIC ANALYSIS PLOTS

This appendix contains network traffic analysis plots for experiments comparing codec types, computer hardware, camera variations, and operating system variations.

The following sequence of events was followed in order to produce the plots in Figures 23 to 28.

Table 3: Order of events for captures

Time	Action
0	Establish call
10	Lights off
15	Light on
25	Lights dimmed to half brightness
30	Light on full
40	Walk past camera
45	Walk back past camera
50	Wave hand past camera
60	Cover camera
65	Uncover camera
75	End call

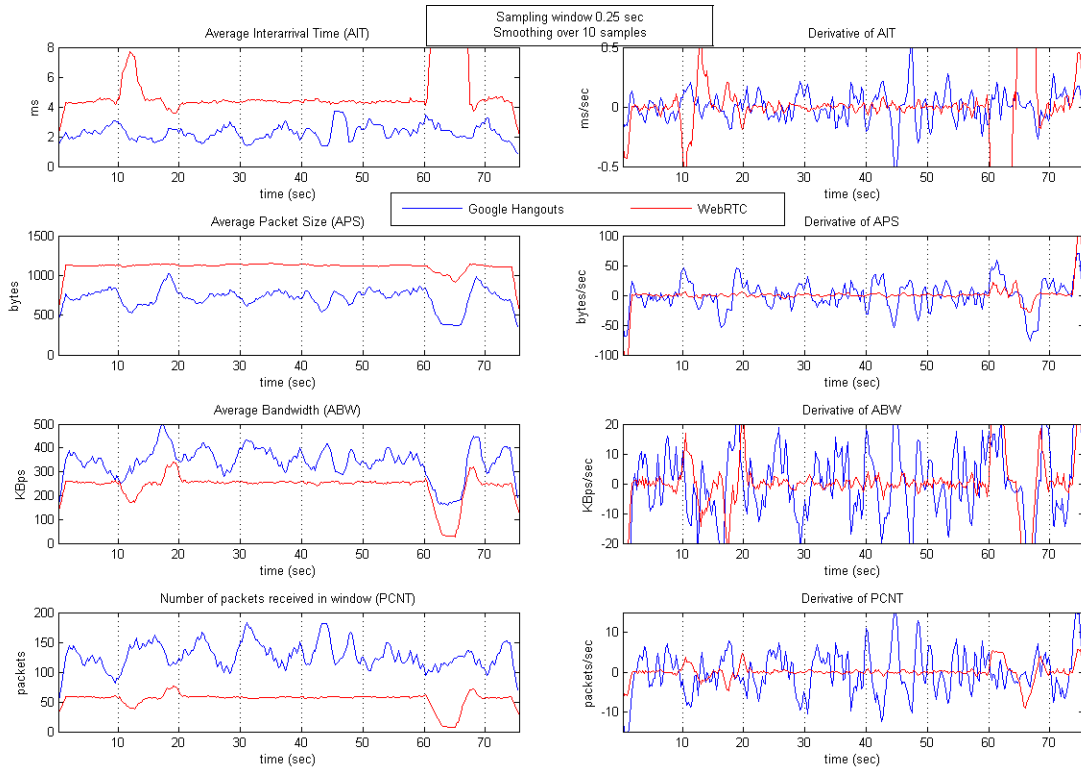


Figure 23: Captures from Google Hangouts and WebRTC.

Figure 23 shows captures obtained from Google Hangouts and WebRTC's [7] demonstration application [8]. Each of these chat services uses an implementation of the VP8 codec.

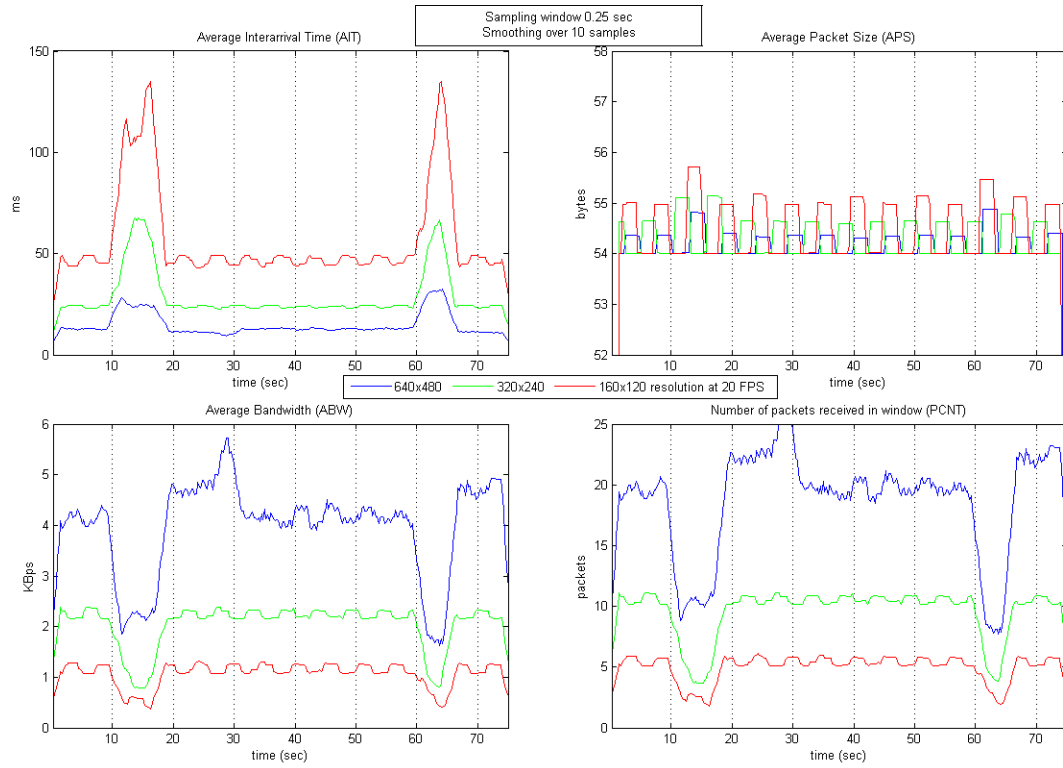


Figure 24: Captures from D-Link camera with varying resolution.

Figures 24 and 25 are all obtained from a D-Link DCS-932L security camera which is encoding using the MJPEG codec.

Figure 24 demonstrates variation in image resolution for each capture. Resolutions shown include 640x480, 320x240, and 160x120 with the encoder set to 20 frames per second for each capture.



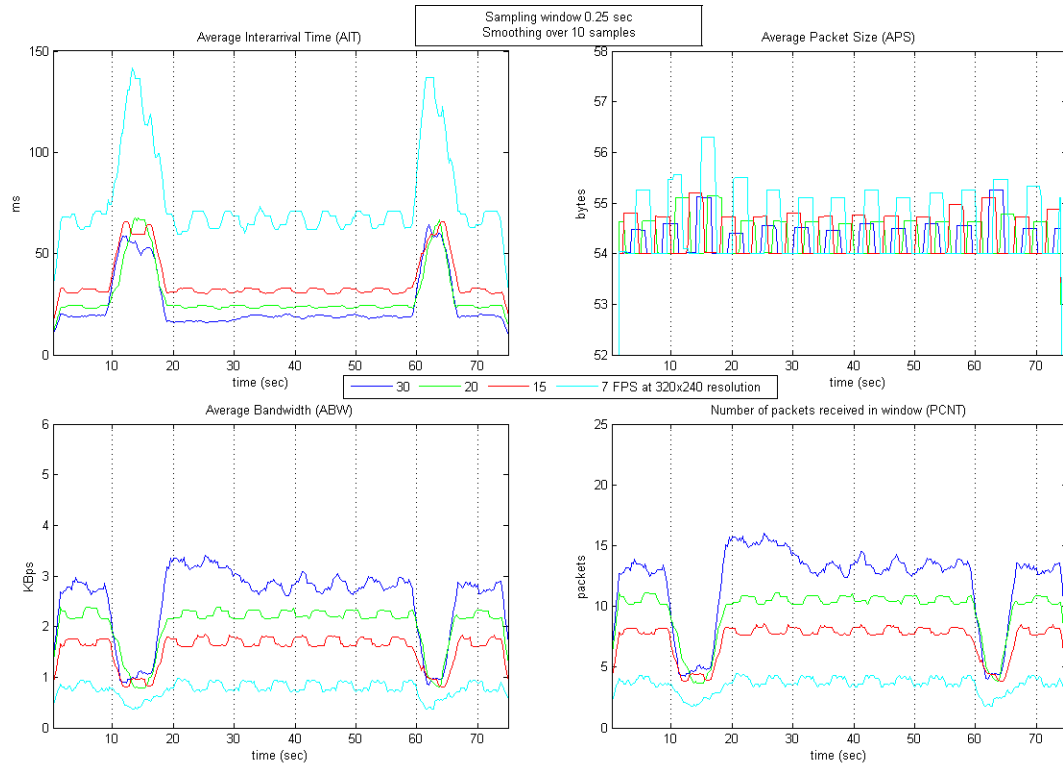


Figure 25: Captures from D-Link camera with varying frame rates.

Figure 24 demonstrates variation in frame rate for each capture. Frame rates shown include 30, 20, 15, and 7 frames per second with the encoder set a resolution of 320x240 for each capture.

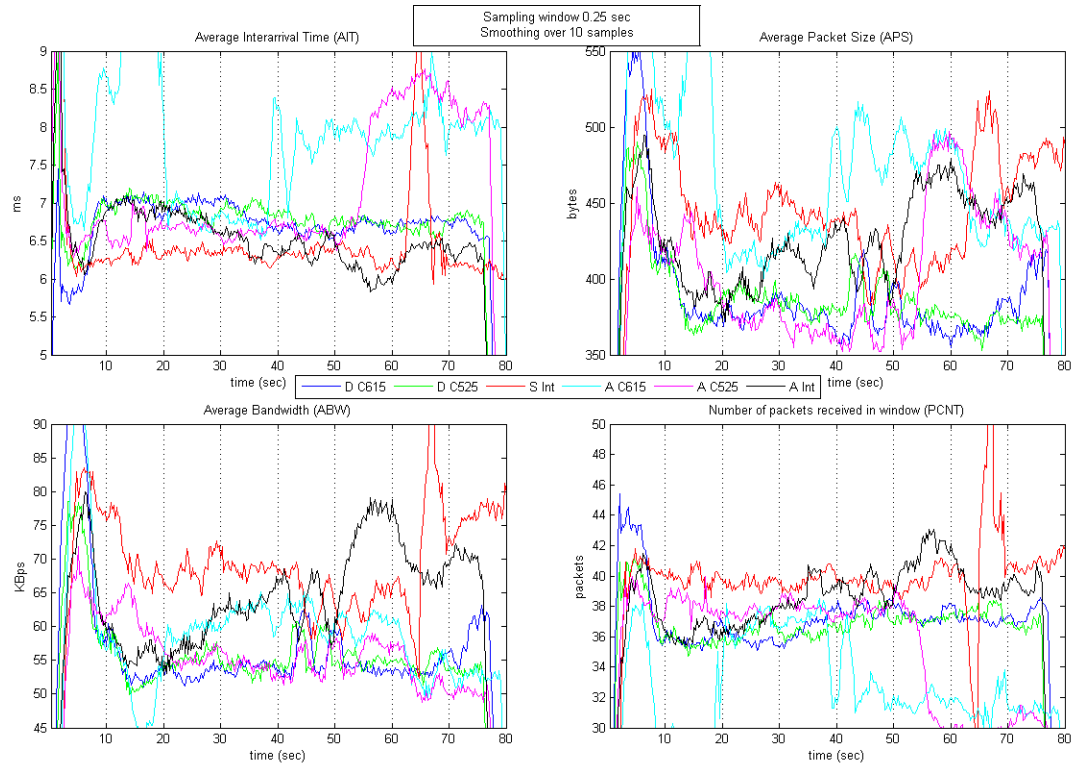


Figure 26: Captures from Dell, Samsung, and Aspire devices with various cameras.

Figures 26 to 28 are all obtained from Skype video chat sessions with the H.264 codec.

Figure 26 shows captures obtained from three devices: Dell Latitude E6530 laptop, Samsung Galaxy S4 smartphone, Acer Aspire One D270-1824 laptop. There are two external cameras used: Logitech C615, Logitech C525. Combinations of external and internal cameras are used with each device and the resulting traffic analysis is shown.

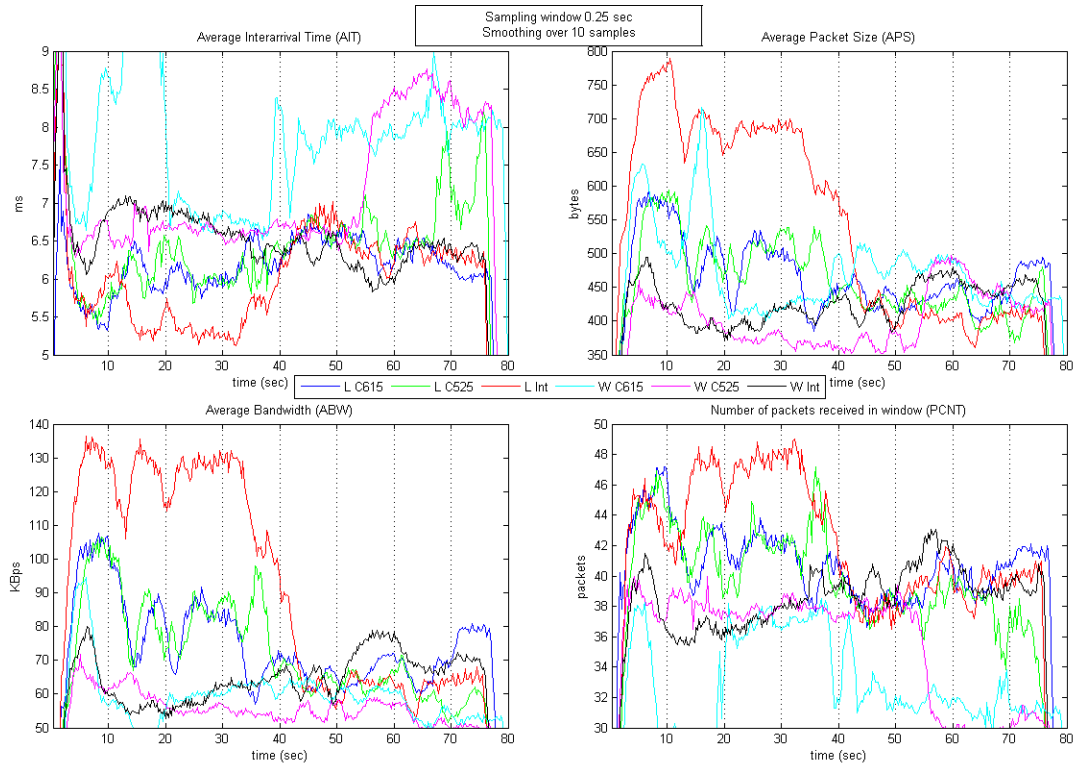


Figure 27: Captures from Acer laptop running Ubuntu 12.04 and Windows 7.

Figure 27 shows captures obtained from the Acer Aspire One D270-1824 laptop. The same cameras are used as in Figure 26. The laptop which dual boots between Ubuntu 12.04 and Windows 7 executed the same sequence of events running Skype under each operating system and the resulting traffic analysis is shown.

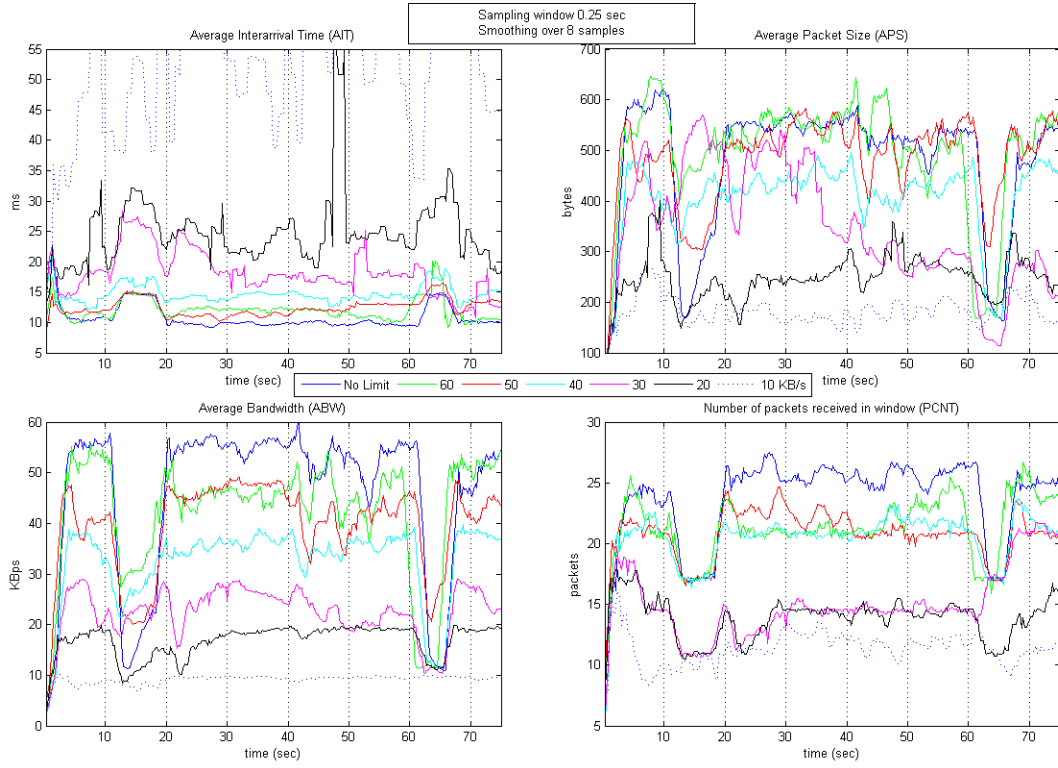


Figure 28: Captures with varying levels of bandwidth limitation.

Figure 28 shows captures obtained by bandwidth limiting the video stream. A base-line capture is shown with no bandwidth limiting applied. In increments of 10 KB/s the bandwidth is then restricted in a range from 60 KB/s to 10 KB/s.

## REFERENCES

- [1] “Advanced video coding for generic audiovisual services.” ITU-T Rec. H.264 and ISO/IEC 14496-10 (MPEG-4 AVC).
- [2] “Skype.” <http://www.skype.com/>, Retrieved Aug 2013.
- [3] “v4l2loopback 0.8.0.” <https://github.com/umlaeute/v4l2loopback/>, Retrieved Jul. 2014.
- [4] “Skype4py 1.0.35.” <https://pypi.python.org/pypi/Skype4Py/>, Retrieved Jun. 2014.
- [5] “Amazon mechanical turk.” <https://www.mturk.com/mturk/>, Retrieved May 2014.
- [6] “Gstreamer - open source multimedia framework.” <http://gstreamer.freedesktop.org/>, Retrieved Nov. 2013.
- [7] “Webrtc.” <http://http://www.webrtc.org/>, Retrieved Oct. 2013.
- [8] “Demo - webrtc.” <https://apprtc.appspot.com/>, Retrieved Oct 2014.
- [9] AIMAR, L., “x264.” [www.videolan.org/developers/x264.html](http://www.videolan.org/developers/x264.html), Retrieved Sep 2013.
- [10] AL-SHAYKH, O. K. and MERSEREAU, R. M., “Lossy compression of noisy images,” *Image Processing, IEEE Transactions on*, vol. 7, no. 12, pp. 1641–1652, 1998.
- [11] BABIKYAN, A., “Sharktools.” <https://github.com/armenb/sharktools/>, Retrieved Aug. 2013.
- [12] BASET, S. A. and SCHULZRINNE, H. G., “An analysis of the skype peer-to-peer internet telephony protocol,” in *the Proceedings of the IEEE Conference on Computer Communications, (INFOCOM)*, 2006.
- [13] BAUCHSPIES, R. A., “Temporal compression and decompression for video,” Dec. 28 1999. US Patent 6,008,847.
- [14] BELLARE, M., KILIAN, J., and ROGAWAY, P., “The security of the cipher block chaining message authentication code,” *Journal of Computer and System Sciences*, vol. 61, no. 3, pp. 362–399, 2000.
- [15] BONFIGLIO, D., MELLIA, M., MEO, M., RITACCA, N., and ROSSI, D., “Tracking down skype traffic,” in *the Proceedings of the IEEE Conference on Computer Communications, (INFOCOM)*, 2008.
- [16] BRUMLEY, D. and BONEH, D., “Remote timing attacks are practical,” *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.
- [17] CHEN, S., WANG, R., WANG, X., and ZHANG, K., “Side-channel leaks in web applications: a reality today, a challenge tomorrow,” in *the Proceedings of the IEEE Symposium on Security and Privacy, (OAKLAND)*, 2010.

- [18] DYER, K., COULL, S., RISTENPART, T., and SHRIMPTON, T., “Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail,” in *the Proceedings of the IEEE Symposium on Security and Privacy, (OAKLAND)*, 2012.
- [19] FABER, V., “Clustering and the continuous k-means algorithm,” *Los Alamos Science*, vol. 22, pp. 138–144, 1994.
- [20] FELLER, C., WUENSCHMANN, J., ROLL, T., and ROTHERMEL, A., “The vp8 video codec-overview and comparison to h. 264/avc,” in *the Proceedings of the IEEE International Conference on Consumer Electronics-Berlin, (ICCE-Berlin)*, pp. 57–61, 2011.
- [21] FRANKEL, S., GLENN, R., and KELLY, S., “The aes-cbc cipher algorithm and its use with ipsec,” tech. rep., RFC 3602, September, 2003.
- [22] HICKMAN, K. and ELGAMAL, T., “The ssl protocol,” *Netscape Communications Corp*, vol. 501, 1995.
- [23] JACK, K., *Video Demystified: A Handbook for the Digital Engineer, 5th Edition*. Burlington, MA: Newness, 2007.
- [24] JACK, K., “Yuv color space,” *Communications Engineering Desk Reference*, p. 470, 2009.
- [25] KOCHER, P., “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *the Proceedings of the International Cryptology Conference on Advances in Cryptology, (CRYPTO)*, 1996.
- [26] MARQUARDT, P., VERMA, A., CARTER, H., and TRAYNOR, P., “(sp)iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers,” in *the Proceedings of the ACM Conference on Computer and Communications Security, (CCS)*, 2011.
- [27] MITROVIC, D., “Video compression,” *University of Edinburgh*.
- [28] ROTTONDI, C. and VERTICALE, G., “Using packet interarrival times for internet traffic classification,” in *the Proceedings of the IEEE Latin-American Conference on Communications, (LATINCOM)*, 2011.
- [29] SURIYANARAYANAN, A., “Inferring video contents using traffic pattern analysis,” tech. rep., Communications Assurance and Performance (CAP) group, Georgia Institute of Technology, 2013.
- [30] VAN ECK, W., “Electromagnetic radiation from video display units: an eavesdropping risk?,” *Computers & Security*, vol. 4, no. 4, pp. 269–286, 1985.
- [31] WALLACE, G. K., “The jpeg still picture compression standard,” *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.
- [32] WELCH, T. A., “A technique for high-performance data compression,” *Computer*, vol. 17, no. 6, pp. 8–19, 1984.
- [33] WRIGHT, C. V., BALLARD, L., COULL, S. E., MONROSE, F., and MASSON, G. M., “Spot me if you can: Uncovering spoken phrases in encrypted voip conversations,” in *the Proceedings of the IEEE Symposium on Security and Privacy, (OAKLAND)*, 2008.